Computationally intensive methods for Hidden Markov Models with applications to statistical genetics

Peter D. Kecskemethy

Merton College, University of Oxford

DPhil Thesis

Department of Statistics, University of Oxford

MT 2014

Dedicated to Laszlo Karpati and Sandor Khoor,
whose example and teachings will never fade

Abstract

In most fields of technology and science, the exponential increase of available data is an apparent trend. In genetics, the main contributor to this trend is the improving efficiency of sequencing technologies. While the Human Genome project focused on assembling a single reference sequence not long ago, now there are aims to sequence million genomes in upcoming projects.

The consequent computational challenge is being able to utilise this wealth of data, which requires the development of sufficiently powerful methods for analysis. However, the speed of transistor-based computing processors has recently hit a power ceiling and developers can no longer rely on hardware improvements automatically providing performance improvements in software directly. The result is that analysis methods are failing to keep up with the speed of data generation, and at this age of exponential data explosion it is becoming critical to find any solution for improving the performance of statistical methods.

One traditional approach is to apply approximations - often trading the quality of results for response time. Another approach is to achieve algorithmic optimisations for existing methods without sacrificing results. Unfortunately, the possibilities for purely algorithmic optimisations often tend to be limited. A third approach is to attempt to harness the computational power of the presently re-emerging field of parallel computing. While the theoretical performance of parallel platforms roughly follows Moore's law, exploiting the power of parallelisms requires significant effort during development and may not even be possible in certain applications.

This work attempts to explore avenues for achieving high performance for Hidden Markov Models (HMMs) and HMM applications in population genetics.

The second chapter of this thesis introduces a single-locus variant of the IMPUTE2 method for calling and phasing genotype variants based on genotype likelihood data. This method uses both approximations and algorithmic optimisations and achieves performance improvements without a considerable drop in accuracy. It is also aimed to be highly parallelisable.

The third chapter presents GPGPU-focused parallelisation methods over the state-space for HMM algorithms specifically under the Li and Stephens model, which is a widely and successfully used approximation of the coalescent. Practical experiments show $\times$200-$\times$6000 times acceleration with a CUDA implementation of the popular Chromopainter method, which is based on the Li and Stephens model.

The last chapter explores the theoretical possibility of parallelising HMM algorithms across blocks of observations (inspired by but not limited to methods used in genetics). A novel view and derivation is presented for block parallelism, along with accompanying analyses of applicability and relevance. Performance analysis results indicate that the application of block-parallelism is expected to be highly relevant for most large-scale HMM applications on present-day computing platforms, while block-parallelism may become a necessity for utilising the improving power of parallel hardware in the close future.

ACKNOWLEDGEMENTS

First of all, I would like to thank my supervisor Chris Holmes, whose example, positivity, enthusiasm for science and trust in me was instrumental for my success.

I would like to thank the Doctoral Training Center and Merton College for their support, and especially Elspeth Garman and Simon Jones, whose support and generous kindness helped me through the hardest times.

I would like to thank everyone who supported me through discussions and helpful criticism, among them Michalis Titsias, Istvan Reguly, Mike Giles, Luke Cartey, David Gavaghan, Charlotte Dean, Jotun Hein, Gerton Lunter, Gil McVean and Simon Myers. My special thanks go to Chris Yau who has my greatest respect as a researcher.

Thanks to all those who made my time in Oxford fun and delightful, Madeline Mitchell, Jeremy Becker, Quin Wills, Janet Gooi, Mireille Gomes, Marzyeh Ghassemi and Tony Haines.

Thanks to all those who helped me start on this path, including Peter Antal from BME, Graham Megson, Chris Guy, Slawomir Nasuto and Muniyappa Manjunathaiah from Reading and Sandor Khoor. Without them I wouldn't even have started.

I would like to thank those, who have helped me personally through both happy and dark times. Firstly, my mom Edith Karpati who always tried to push me in the right direction and my grandfather Laszlo Karpati who had been the best possible example to be had. My never-fading gratitude goes to Reka Balogh who was my main support and other half for many tough years under all circumstances. My warmest thanks go to Emma Bornai for her friendship with a depth unparalleled. Thanks to all my good friends at home who always welcomed me back. Thanks to my brother Zoltan Kecskemethy for always managing to demonstrate a better version of me and thanks to Annie Ng who was there with me in the end.

Finally, I would like to acknowledge Jonathan Marchini, for without his counterexample I would have not seen important limitations of academia, for without his counterexample I would have remained in a field not suitable for me and for without his influence on me and my work I would not have grown as strong and independent as I have.

CONTENTS

Chapter I

BACKGROUND

1 Hidden Markov Models and the Li and Stephens model

1.1 General Hidden Markov Models

Hidden Markov Models (HMMs) are versatile generative statistical models widely used across a multitude of fields in science and engineering for temporal and spatial prediction and pattern recognition problems. Application areas range from speech recognition, computer vision and finance to artificial intelligence and genetics (Aas et al., 1999; Li et al., 2009; Rabiner, 1989; Yau and Holmes, 2010). In particular, HMMs are arguably the most widely used probability models in bioinformatics.

There are various types of HMMs and multiple different approaches of describing them. In essence, HMMs are generalised Markov chains (MCs) with unobserved states (which may refer to classifications of loci in sequences, for instance) that generate observations (emissions) probabilistically. HMMs have a (possibly infinite and possibly unknown) number of states with a transition probability distribution which characterises transitions between them, similar to that in ordinary MCs. In standard HMMs each state is further associated with an emission probability distribution that defines the probabilities of generating possible emissions form an emission alphabet given the current state of the HMM. The probabilities of starting in any of the states is defined by an initial state distribution (Rabiner, 1989).

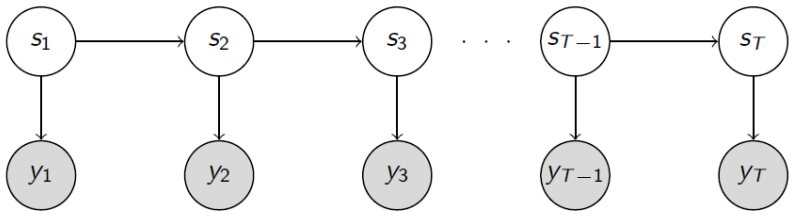

Figure I.1: HMM as a (directed) graphical model. Arrows represent conditional dependencies and the absence of arrows represent conditional independencies.

More formally

A discrete-state discrete-emission HMM is defined by the possible states $\mathcal{S}$, the possible emissions alphabet $\mathcal{E}$, the emissions probabilities $\mathrm{P}(\, y \in \mathcal{E} \mid s \in \mathcal{S}\,)$, the state transitions $\mathrm{P}(\, s_t \mid s_1, ..., s_{t-1}\,) \;=\; \mathrm{P}(\, s_t \mid s_{t-1}\,)$ with Markov property and the initial distribution π over $\mathcal{S}$.

The Markov property that the future $S_{t+1}, ..., S_T$ does not depend on the past $S_1, ..., S_{t-1}$, given the present S_t is key to the success of HMMs. This conditional independence may be explained through a graphical model representation where the hidden state is considered as a different variable over time (Figure I.1). The corresponding simplified joint distribution for HMMs can be written as

$$\mathrm{P}(y_1, ..., y_T, s_1, ..., s_T) \;=\; \mathrm{P}(s_1)\,\mathrm{P}(y_1 \mid s_1) \prod_{t=2}^{T} \Big[\, \mathrm{P}(y_t \mid s_t)\,\mathrm{P}(s_t \mid s_{t-1}) \Big].$$

Computation

Several standard HMM algorithms have been developed to solve the computational problems related to HMMs. The forward algorithm is used to calculate the probability $\mathrm{P}(y_1, ..., y_T \mid \lambda)$ of a given observation sequence under a fixed parameterisation. The Viterbi algorithm is used to find the most probable state sequence(s) $\arg\max \mathrm{P}(s_1, ...s_T \mid y_1, ...y_T, \lambda)$ that could have resulted in a given sequence of observation. The forward-backward algorithm is used to calculate the probabilities of being in each state at each point given an observation (i.e. the marginal probabilities $\mathrm{P}(s_t \mid y_1, ..., y_T)$ for each state and t). Finally, the Baum-Welch algorithm is used to adjust (i.e. learn) the model parameters to maximise the probability of a set of observations. All these algorithms apply dynamic programming to exploit the Markov property (see Figure I.2) and achieve a computational complexity $\mathcal{O}(N^2T)$ linear with the observation length (Rabiner, 1989).

The theory of HMMs is highly developed and many HMM algorithms are modified for domain-specific use. Unfortunately the computational requirements are still relatively high when applied to large datasets (especially when N is large) and may require long computation times in practice, which is especially relevant in population genetics.

1.2 The Li and Stephens model

The Li and Stephens (LS) model has been one of the most widely used models in population genetics since its development (Li and Stephens, 2003). It is essentially a model of linkage disequilibrium (LD) in the form of a conditional sampling distribution (CSD), which serves

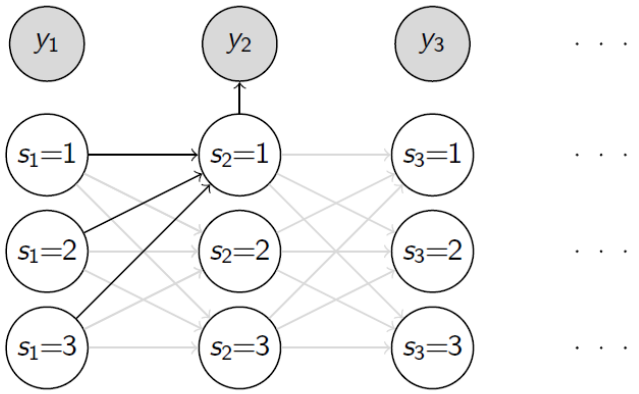

Figure I.2: HMM computation. The figure demonstrates what may be called a single step in the standard dynamic programming-based algorithms (here the forward algorithm) at t=2 for state 1 for an HMM with 3 states.

as a computationally efficient approximation to the coalescent with recombination (Hudson, 1983; Kingman, 1982).

The LS model allows for the development of efficient methods of inference for a variety of problems in population genetics, such as calculating recombination rates, phasing, imputing haplotypes and genotypes and determining population structure and population history. The development of the LS model and similar MC-based models was a considerable breakthrough in population genetics and often they still provide the starting or reference point for developing new methods (PHASE (Stephens and Scheet, 2005), fastPHASE (Stephens and Scheet, 2006), IMPUTE (Marchini et al., 2007), IMPUTE2 (Howie et al., 2011, 2009), MACH (Li et al., 2010) and others (Lawson et al., 2012; Paul and Song, 2012); etc.).

The original Li and Stephens (2003) paper describes the LS model with constant recombination rates and a single-hotspot model. It also discusses a way to correct for bias by altering the transition matrices. The model description below and all algorithms and implementations in this work assume a slightly altered version of the LS model, where the recombination rates can vary across the genome and in which the bias-correction is not included.

The LS is a framework for modelling 'inference' or 'recipient' haplotypes $\mathcal{H}' = \{\boldsymbol{h_1}, .., \boldsymbol{h_K}\}$ conditional on a set of 'reference' or 'donor' haplotypes $\mathcal{H} = \{\boldsymbol{H_1}, .., \boldsymbol{H_N}\}$, both defined on the same set of loci $l \in \{1, ..., L\}$. Recombination is modelled by each haplotype $\boldsymbol{h_k}$ copying sections from the haplotypes in $\mathcal{H}$. When copying occurs between $\boldsymbol{h_k}$ and $\boldsymbol{H_n}$, it is thought to suggest possible recent shared ancestry between $\boldsymbol{h_k}$ and $\boldsymbol{H_n}$. The equivalent of this copying mechanism in coalescent models is that the two haplotypes are on the same branch somewhere in the population's genealogy with respect to a shared section $[l_a..l_b]$ of the sequence. This copying mechanism models past recombination events in the population.

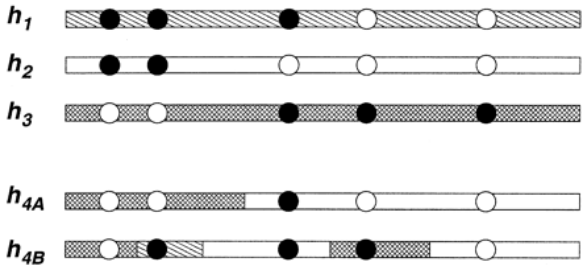

Figure I.3: Modeling sequences as an imperfect mosaic of other sequences in the LS model (Li and Stephens, 2003). On this illustration haplotype h_4 is modelled two ways as imperfect mosaics of haplotypes h_1, h_2 and h_3. Shading shows which of haplotypes $h_1 ... h_2$ are copied at which position.

Importantly, the copying in the LS model can be imperfect, allowing for (usually minor) dissimilarities between $\boldsymbol{h_k}$ and $\boldsymbol{H_n}$ and, thus, effectively modelling mutation events. This 'imperfect mosaic' modelling of haplotypes is illustrated in Figure I.3.

The above imperfect copying mechanism is formulated as a non-homogeneous MC with probabilistic emissions. In the LS model the states s of the MC correspond to each of the reference haplotypes $\boldsymbol{H_s}$, where the MC being in state s at the current locus corresponds to the current inference haplotype $\boldsymbol{h_k}$ copying from $\boldsymbol{H_s}$ at the locus. The transition probabilities of the MC can vary along the genome, determined by local recombination rates, while the emissions are constant and are only of two types corresponding to mutations and non-mutations (see eq. (1.2)).

It is clear that HMM algorithms can be applied to the LS model. Importantly, a simplification in the transition structure of the LS model allows for the development of model-specific HMM algorithms that can achieve linear computational complexity (in both the number of states as well as the length of observations). The simplification is that transitions from a state s_l to a different new state $s_{l+1} \neq s_l$ are equally probable at a given locus, making the state changes independent of the present state.

$$\mathrm{P}(\, s_{l+1} \mid s_l \,) = \begin{cases} e^{-\rho_l} + \dfrac{1 - e^{-\rho_l}}{N} & \text{when } s_{l+1} = s_l \\ \dfrac{1 - e^{-\rho_l}}{N} & \text{otherwise} \end{cases} \tag{I.1}$$

with

$$\rho_l = \frac{4\, c_l\, d_l\, N_{eff}}{N},$$

where $c_l\, d_l$ is the genetic distance between loci l and $l{+}1$. The constant N_{eff} is the effective (diploid) population size and the factor c_l is the average rate of crossover per unit physical (base pair) distance per meiosis between sites l and $l{+}1$ measured in Morgans.

The transition probabilities can be interpreted to mean that a $s_l \to s_{l+1}{=}s_l$ transition can happen two ways - the Markov chain stays in the same state (with 'staying probability' $e^{-\rho_l}$) or happens to switch into the same state out of all the N possible states. The simplified transition structure also means that at each locus, all transition probability distributions can be fully characterised by the 'staying probabilities' $\mathrm{P}(\, s_{l+1}{=}s_l \,|\, s_l \,)$ and the number of states N alone. Li and Stephens noted how the simplified structure allows algorithmic simplifications and acceleration to $\mathcal{O}(LN)$ complexity (see (Li and Stephens, 2003) and Chapter III in this work). This structure is also useful for minimising memory requirements that tend to be high in HMM implementations (again, see Chapter III for details).

The emission probabilities of the LS model are given as:

$$\mathrm{P}(\, \boldsymbol{h_k}[l] \mid s_l \,) = \begin{cases} 1-\mu & \text{for no mutation } (\boldsymbol{h_k}[l] = \boldsymbol{H_s}[l]) \\ \mu & \text{for mutations } (\boldsymbol{h_k}[l] \neq \boldsymbol{H_s}[l]) \end{cases}$$

where $\boldsymbol{h_k}[l]$ is the allele of haplotype $\boldsymbol{h_k}$ at locus l, $\boldsymbol{H_s}[l]$ is the allele of the reference haplotype $\boldsymbol{H_s}$ corresponding to state s at locus l, and μ is either given or fixed at

$$\mu = \frac{\bar{\theta}}{2(\bar{\theta} + N)} \tag{I.2}$$

using Watterson's estimate

$$\bar{\theta} = \left(\sum_{m=1}^{N-1} \frac{1}{m} \right)^{-1}.$$

2 Application areas in population genetics

2.1 The importance of studying genetic variation

The study of genetic variation is an important pillar of the advancement of modern biology. It provides a technology-driven 'fast track' to explore cell mechanisms and identify areas for research.

Our knowledge of genetics has significantly increased in recent years. Following the success of various large-scale association studies, there are now over 1000 single nucleotide polymorphism (SNP) variants that have been found in humans to have effect on more than 200 phenotipic traits, including diseases such as schizophrenia, various types of cancer and diabetes, as well as on quantitative physical characteristics and conditions like height and obesity (Institute, 2011; Manolio, 2010). These discoveries are important for multiple reasons. Discovering the polymorphisms that take part in the development of diseases is an essential part in understanding the Biology behind them. As the past has shown in many other areas of human biology (e.g. the study of the human brain), the understanding of diseases motivates research on the normal operation of biological processes by giving clues, starting points and footholds (for instance about which genes may have a role in a certain cell mechanism) for hypothesis generation. The improved understanding of diseases and the general advancements of biology together are important for supporting drug design, treatment development, the possibility of personalising healthcare and the assessment of risk one has to a given set of diseases.

2.2 Genome-wide association studies and genetic markers

When exploring the genetics of a phenotype in question, a usual first step is to find associations between genotypic and phenotypic variations. Genome-wide association studies (GWAS) look at large numbers (in millions) of genetic variations, called markers, both causal (responsible for causing the phenotypic differences in question) and non-causal.

Most known causal genetic variations are located in genes, but it is suspected that they can also be found in regulatory elements outside genes, such as promoter regions and splice sites. Most genetic variation has only a small effect on phenotype, but collectively they can have a large effect due to both cumulation and interactions between them. The power of statistical methods by which genetic data are analysed enables researchers to find variations that only have a small effect on the phenotype, for example increasing disease susceptibility only by say 10%.

Due to the complex correlation structure between different parts of the genome (called Linkage Disequilibrium or LD), markers can guide exploration even if causal variations are absent from the GWAS study data. Non-causal sites can serve as useful markers: associations found with any marker can highlight certain parts of the genome that are in high LD with it (for instance the area around it) as candidate regions for further study (Mills et al., 2006). With the spread of cheaper next-generation sequencing technologies the sequencing of whole genomes is becoming more available, in which case causal variants may be found directly.

2.3 Expanding methods to structural variants

GWASs using SNPs have been established as powerful tools and have been widely used in the past, while the study of other types of genetic variation has drawn significantly less attention (Albers et al., 2010). Gradually, it has been recognized that other types of genetic variation, such as copy number variations (CNVs), short insertions and deletions (indels), microsatellites and large structural variations may also serve as useful markers (Vali et al., 2008).

The imbalance in genetic variation studies towards SNPs and large-scale structural variations can be attributed to the field's dependence on the availability and affordability of sequencing and genotyping technologies. With the advent of wide-scale use of so-called 'next-generation sequencing' techniques and their use in the 1000 Genomes Project (1000GP, The 1000 Genomes Project Consortium (2011)), it became possible to explore human genetic variation far beyond the realm of SNPs. The resolution and amount of data produced in the 1000GP allows the identification of indels and other types of structural variants, facilitating their analysis along SNPs. While indels are only about one-eighth of SNPs in abundance, it is reasonable to think that indels can serve as markers similarly to SNPs and that individual indels can have stronger functional relevance than individual SNPs, for instance due to their ability to produce frameshifts in coding regions.

2.3.1 Indels

Short insertions and deletions form an important subset of genetic polymorphisms. There is about one indel for every eight SNPs in the human genome, and so indels are less frequent, but have a great functional significance (Cartwright, 2008; Lunter, 2007a).

Indels have been associated with many human pathologies such as cystic fibrosis, various types of cancer and neurological diseases (Christopher E. Pearson and Cleary, 2005; Duval and Hamelin, 2002; et al., 2006; Garcia-Diaz and Kunkel, 2006).

Insertions and deletions can occur in both coding and non-coding regions of the genome. Since the amino acid of protein products are coded by base triplets, any insertion or deletion of lengths that are not multiples of three cause frameshift mutations by changing the frame of interpretation of the whole sequence after such an indel. Although this can be beneficial (for example in microbe populations), most frameshift mutations are deleterious and occur rarely in coding regions. In non-coding regions however, indels can be of any length, and indel frequencies have an inverse relationship to their length. Indels that cause no frameshift mutations in genes and indels in non-coding DNA regions are also known to have severe effects. Even a single extra or missing base pair can have a great effect on the regulatory functions of non-coding regions, such as splice sites, transcription factor binding sites, recombination hotspots or TATA boxes[1]. In fact, indels can have an effect on most biological process that involve motif recognition (The 1000 Genomes Project Consortium, 2010).

Insertion and deletion mutations appear mostly as results of serious errors in the DNA replication pipeline of cell nuclei. In order for DNA replication to take place, double stranded DNA is first unwound by a topoisomerase and opened by the DNA helicase protein. Different DNA polymerases bind to their appropriate strands of the then accessible single strands of DNA. The polymerases then essentially synthesise complement strands of DNA with the assistance of a range of other different proteins (in slightly different fashions on the two strands) by 'crawling through the region'. The process normally results in two identical DNA double helices in which the nucleotides of opposing strands are all matching (Alb, 2008; Pray, 2008).

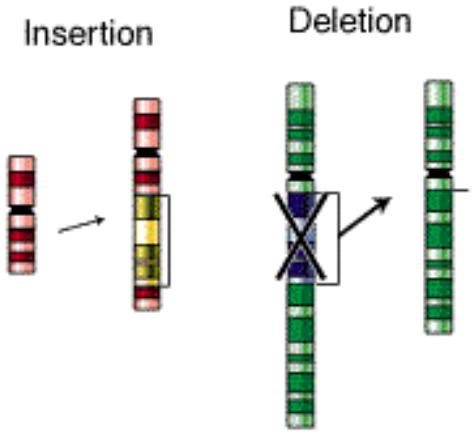

Figure I.4: Diagrammatic explanation of Insertions and deletions. Despite the scale suggested by the diagram, the focus in this work is on short indels of about 1-50 base pairs.

It is the various faulty behaviours of the different DNA polymerase complexes that are thought to be responsible for most small deletion mutations. DNA replication is a robust machinery augmented with many error correction mechanisms. In vitro studies reported the error rates[2] of 0.06×10^{-5} for deletion and less than 0.01×10^{-5} for insertions, but these figures largely vary for the different polymerases and tend to be higher in repetitive regions of sequences (Garcia-Diaz and Kunkel, 2006). Essentially, many things need to go wrong in the

[1] A TATA box is a motif consisting of a sequence of thymine and adenine nucleotides TATAAA. TATA boxes promote transcription of genes which they precede (Alb, 2008).

[2] Error rate: number of errors per nucleotide synthesized

DNA replication process for indels to become permanent mutations. The polymerase needs to make an error, for example sliding (slipping) one base pair further forwards or backwards on the single stranded DNA, while the polymerase needs to remain catalytically operable, and the mistake can not be corrected or needs to be corrected incorrectly by genetic post processing.

In the case of deletions, the molecular basis of occurrence is generally down to the 3D structure of the DNA-polymerase complexes as follows: after the fault of producing a loop on the template strand, the replication sometimes manages to be continued when the complex remains stable and catalytically operable. The stabilisation of the complex with the extra loop can be caused by various perturbations including extrahelical nucleotide intermediates, the misalignment of dNTP substrates, 'active-site misalignment' induced by disturbing conformations of the template strand or the copying of damaged DNA. These events are more likely to occur at the replication of repetitive regions because similar neighbouring sequence parts facilitate the use of intermediates, which can satisfy stability criteria when a slippage occurs (Garcia-Diaz and Kunkel, 2006; Pray, 2008).

DNA polymerases can detect geometric distortions caused by mutations by monitoring correct base pairing through hydrogen bonds and van der Waals contacts with the bases. The efficiency of these proofreading mechanisms again depends on the sequence. The correction mechanism is very robust for short repetitive or non-repetitive sequences, but its efficiency diminishes with the number of repetitions. Normally after replication, cells have further machinery to correct mutations, one of which is called post-replication mismatch repair. This function is crucial for correct DNA biosynthesis and is a major factor in removing indel mutations. Without mismatch repair, indels remain repetitive sequence elements, leading to microsatellite instability, which is thought to precede the development of cancer. Many times it is the fault (e.g. absence) of correction methods that are indirect causes of indel mutations, but sometimes it is the incorrect behaviour of correction mechanisms that turn out to be direct causes of mutations (Garcia-Diaz and Kunkel, 2006).

While there have been many studies confirming hypotheses (such as Streisinger's 'strand-slippage hypothesis') on the reasons for single deletion mutations, the mechanisms that result in insertions or longer indels remain largly less known. On the whole, indel formation is far less understood than the biology of SNP mutations (Garcia-Diaz and Kunkel, 2006).

3 Parallel computation

In the past decades the development of ever increasingly complex computational methods (and software in general) depended on the assumption that the processing power of computer

hardware will continue to steadily increase according to Moore's law (Moore, 1965). While the increase in the number of processing components has been following this trend, the speed improvement of modern processors hit a power ceiling around 2006. The result is an increased number of components at relatively constant (or rather slightly decreased) clock speed, which makes it clear that in the future software development can not rely simply on advances in hardware speeds 'hidden' from programmers.

While processor speed has become relatively stagnant, the rate of data generation is accelerating in most fields and modern software have ever increasing speed requirements. This trend is especially apparent in genomics, with a clear exponential increase in the amount of data produced. Since the human Genome Project and the Hapmap Project, the 1000 Genomes Project has become a reality and large-scale sequencing projects have been announced with increasing sizes (UK10k Human Genome Project, 100,000 Genomes Project and The Million Human Genome Project).

In order to be able to cope with modern computing needs, the trend of developing parallel hardware structures reappeared in recent years (Esmaeilzadeh et al., 2012; Kirk and Hwu, 2010; NVIDIA, 2014). Parallel computing is not a novel field, although it has been in rapid expansion since its recent revival. The world of parallel architectures currently includes distributed computing, central processing unit-based (CPU) multi-core parallelism and graphical processing unit-based many-core architectures as well as less flexible approaches, such as field-programmable gate array (FPGAs). Each architecture has very distinct features, advantages and disadvantages.

The inherent performance limitation of distributed systems is the high cost of communication between distant nodes. Distributed parallelism is therefore only applicable in situations where problems are reducible to completely independent sub-tasks with no need for communication between running processes.

CPU-based supercomputers are currently the most general-purpose parallel architectures in existence. They dominate the field of high-performance computing, making up most of the world's top 500 supercomputers, and are capable of achieving the highest peak performance (up to 80% computing capacity - `www.top500.org`). CPU cores are versatile - they have the advantage of being able to execute code efficiently even when programs have complex flow. While most modern CPUs consist of multiple cores, the downside of CPUs is that they require complex control logic, which makes them expensive and relatively complicated to put together into parallel systems. Large-scale CPU-based parallel systems are very expensive, require large physical space, a lot of energy and high maintenance. They also have relatively low performance in terms of inter-process communication, which makes their performance limited in many applications.

Figure I.5: GPUs devote more transistors to data processing but have significantly less control logic (NVIDIA, 2014).

The future of both affordable and high-end supercomputing may well lie in the use of graphical processing units (GPUs) and hybrid (CPU-GPU) systems (Nickolls and Dally, 2010; Owens et al., 2008; Vuduc and Czechowski, 2011). GPUs are massively parallel, single instruction multiple threads (SIMT) devices with a large number of simplified compute cores on a single chip (see Figure I.5). They are optimised to execute the same task a huge number of times as well as to access memory efficiently. The general purpose versions (GPGPUs) are currently the obvious and only available choice of high-performance computing (HPC) hardware for general users. GPUs have a low power consumption per computation ratio and are affordable (Nickolls and Dally, 2010; Owens et al., 2008; Vuduc and Czechowski, 2011). With the design of having groups of many closely coupled computational cores on a single chip, GPUs provide hardware facilities to minimise the communication cost between parallel threads (within some limits). See a simplistic baseline comparison of CPUs with GPUs in Figures I.6 and I.7. Currently, the use of GPUs is expanding to such an extent that GPUs are even making their way into low-power portable devices, such as mobile phones.

Some suggest that the usefulness of multi-core systems may plateau in the near future (Bordawekar et al., 2010; Esmaeilzadeh et al., 2012; Lee et al., 2010; Vuduc et al., 2010), but it is likely that balanced hybrid architectures will prolong their usefulness (Owens et al., 2008; Vuduc and Czechowski, 2011). Besides the explosion in the development and production of GPGPUs, CPUs are also becoming more and more parallelised, with an increasing number of cores and with the improvements of internal vector processing. This is an obvious alignment in the direction of CPU and GPU development. The development of multiple hybrid architectures have been announced by major manufacturers (including AMD's heterogeneous uniform memory access architecture), and Intel has already shipped Haswell processors with integrated Iris Graphics (including GPU memory). There have also been patents filed for hybrid co-processors that also allow for hope that there will be HPC-ready hybrid hardware

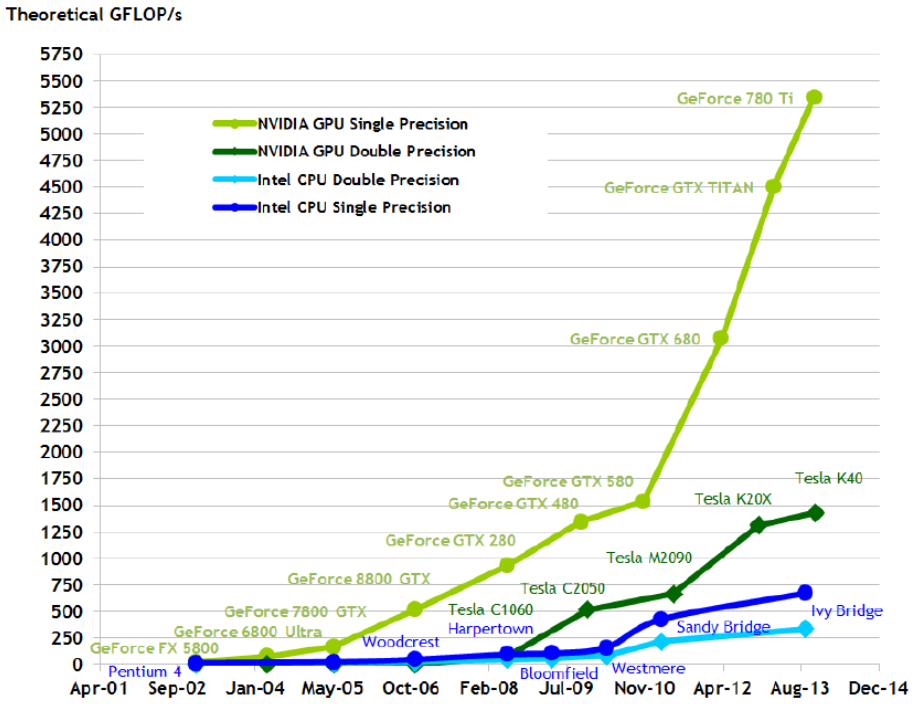

Figure I.6: Floating-Point operations per second for CPUs and GPUs over recent years (NVIDIA, 2014).

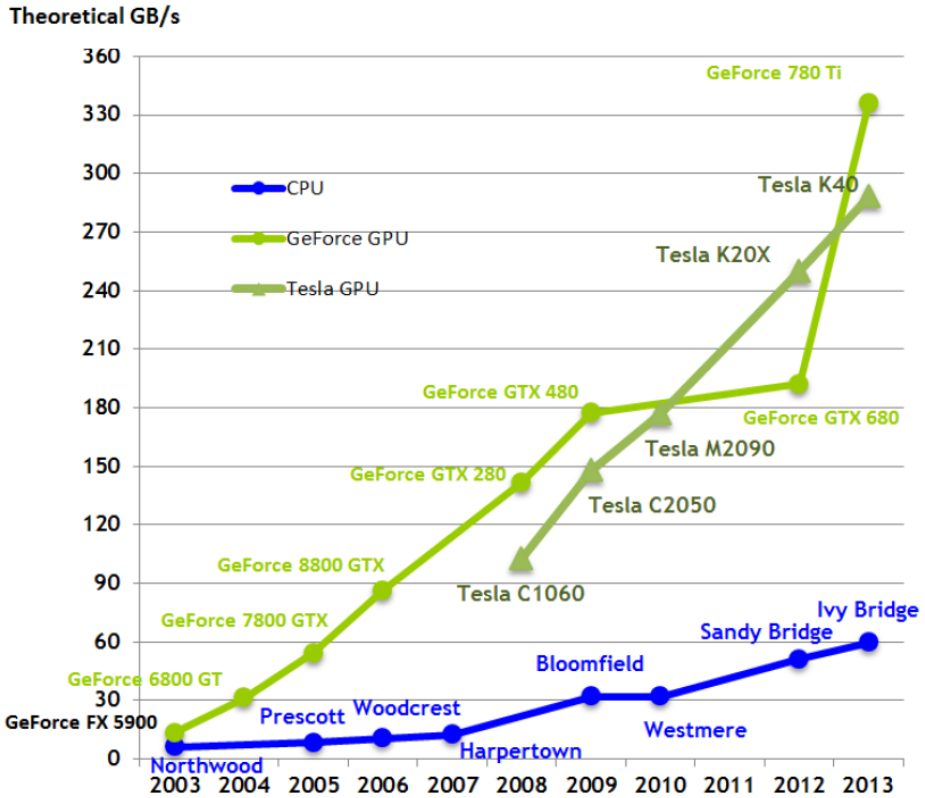

Figure I.7: Memory bandwidth improvements for CPUs and GPUs over recent years (NVIDIA, 2014).

available in the close future.

The most important improvement we can expect from hybrid systems is that host-GPU communication will be more efficient (with higher bandwidth and lower latency), which is currently a strong performance and development limiting factor in many applications.

3.1 General purpose graphical processing units (GPGPUs)

A small number of suppliers dominate the production and development of the GPU market, essentially making GPU-based HPC closely tied with a few dominating companies and their proprietary approaches. One of the commercial leaders is NVIDIA, a pioneer in GPU manufacturing and a dedicated supporter of GPU-based HPC. In fact it was the appearance of NVIDIA's GPGPUs and its proprietary CUDA language that gave the GPU-based HPC field its real start (Kirk and Hwu, 2010).

Long after its launch, CUDA still dominates the field of high performance GPU programming. Currently the only significant competitor is the open OpenCL language, which is partly inspired by CUDA in its design. OpenCL is a standardised language and its development is currently falling behind CUDA both in terms of functionality and efficiency.

Low-level (meaning close-to-hardware or "hands on") parallel programming can require significant human resources and so it is often advisable to look at alternative approaches. The programmer can either 1) formulate his computational task as a matrix algebra problem and use a linear algebra libraries that makes use of GPU resources; 2) use some other domain-specific library that can make use of GPU resources, if applicable; or 3) use a higher level framework (such as OpenACC) for potentially less efficient programming. The optimal choice depends on the application in question.

3.1.1 CUDA

CUDA is an extension to the C programming language (there also exists a version for FORTRAN and wrappers for certain other languages such as Python), specifically designed for parallel computing on various devices of NVIDIA (NVIDIA, 2014). Although CUDA is currently the most widely used language for GPU programming and the most prominent driver of GPU computing, its development is still in a dynamic stage. CUDA experiences considerable changes reflecting the advancements in each generation of graphics devices (Owens et al., 2008). CUDA is a relatively low-level language, partly forcing the programmer to constantly think about the hardware, but also allowing the programmer to have a direct, relatively low-level control over computational resources, which greatly enables optimisations.

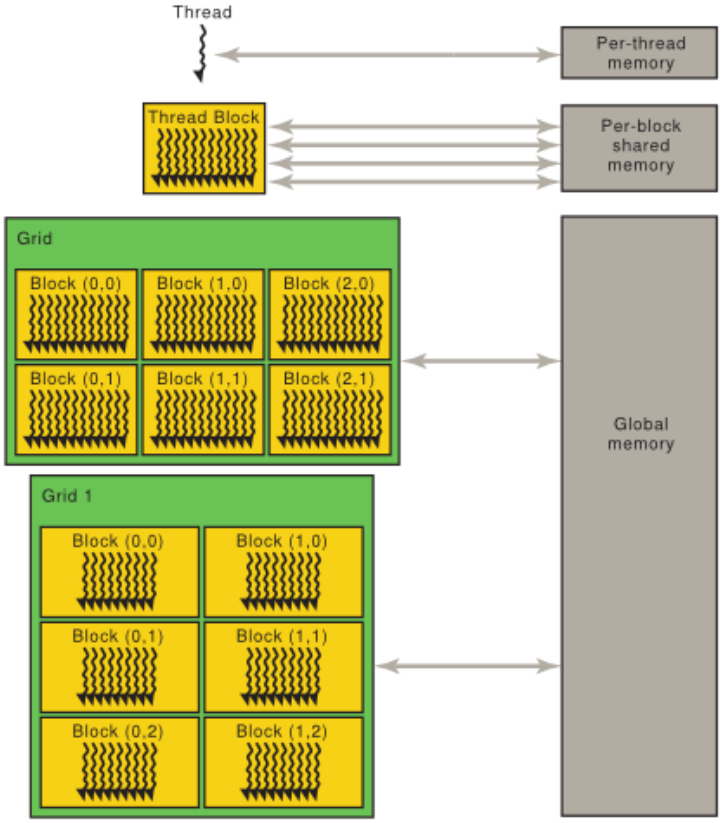

Figure I.8: The CUDA programming model and memory access (NVIDIA, 2014). Per-thread memory means registers and spilled registers. Constant memory is accessible at every level.

In general, the strength of CUDA is in its simple and powerful abstraction that unifies parallel processes and pieces of hardware under a single hierarchy (see Figure I.8). The different hierarchy levels, devices, grids, blocks and threads, have different expected behaviours and are suitable for different programming tasks (Kirk and Hwu, 2010; Nickolls et al., 2008; NVIDIA, 2014). The lowest level of parallelisation that CUDA provides is over threads. This level is suitable for the parallel execution of tasks that are highly linked in terms of data use and require strong inter-thread communication. This communication can be achieved with an efficiency that is unique to GPUs and provides a critical advantage to GPUs over alternatives. Threads are executed in groups called warps (16-32 threads in each warp on current architectures).

The level of parallelisation over blocks is suitable for independent tasks that require no communication. It is possible to implement information sharing between blocks but such approaches require global synchronisations and tend to be very inefficient even with the use of specifically designed atomic operations provided by CUDA. The largest hardware sub units of NVIDIA GPGPUs (called streaming multiprocessors) can run 2-32 blocks of threads

each in any given time, but the number of blocks defined in a code can be far greater (around $65,536^3$).

The highest level parallelisation is over separate GPGPU devices, which is usually equivalent to that of using separate computing nodes (distributed computing). This setup makes communication and shared data use inefficient but nonetheless provides a valuable and simple top-level possibility for parallelisation (Bordawekar et al., 2010; Nickolls et al., 2008; NVIDIA, 2014).

While the optimal use of different abstraction levels is crucial, CUDA programming requires attention to further details. These details include the explicit and implicit rules and limitations of CUDA, the mechanisms of the underlying hardware architecture, the use of different types of specialised memory types with different access constraints, the optimisation of caching, the distributed CPU-GPU architecture, the concurrency between parallel processes, possible concurrent memory access problems (such as lost updates and necessity locks), synchronisation, branches, deadlocks and the communication between threads (NVIDIA, 2014; Owens et al., 2008; Ryoo et al., 2008). Besides these complications, the debugging of low-level code and parallel processes together make CUDA development a challenging task. An additional difficulty is that CUDA does not allow the calling of standard C functions and the language abstraction of CUDA has not reached a level which allows for object-oriented programming. As a consequence, the development of applications of any size under CUDA has to be done with highly defensive programming - focusing what can go wrong in how many different ways instead of what the potentially simple single-thread process is - in small increments with extensive testing. The downside is that the development process can become somewhat slower and more complex and generally less productive compared to general higher-level programming for CPUs (such as in C++, Java, Python, etc) (Bordawekar et al., 2010; Lee et al., 2010; Vuduc et al., 2010).

Despite the complexities of development in CUDA, it is important to keep in mind that firstly, parallel computing is currently the only apparent way to keep improving performance, and secondly, low-level programming is always the best way towards real efficiency (Ryoo et al., 2008). Moreover, significant improvements can be expected in the abstraction level of CUDA and with the spread of GPU computing, and a rise in the number of readily-usable domain-specific libraries can be expected. Domain-specific libraries are particularly promising as they make use of GPU capabilities while hiding hardware complexities from application programmers.

3.1.2 NVIDIA GPU systems and the CUDA programming model

Beyond the previously mentioned difference that GPUs allocate lower computational power (practically meaning a smaller number of transistors) to control and more to calculations, there are many more important differences between CPUs and GPUs that have practical significance for a developer. The benchmarks and other figures in this section have been taken from (Kirk and Hwu, 2010; NVIDIA, 2014; Owens et al., 2008) and various benchmarks published in the online GPU developer community.

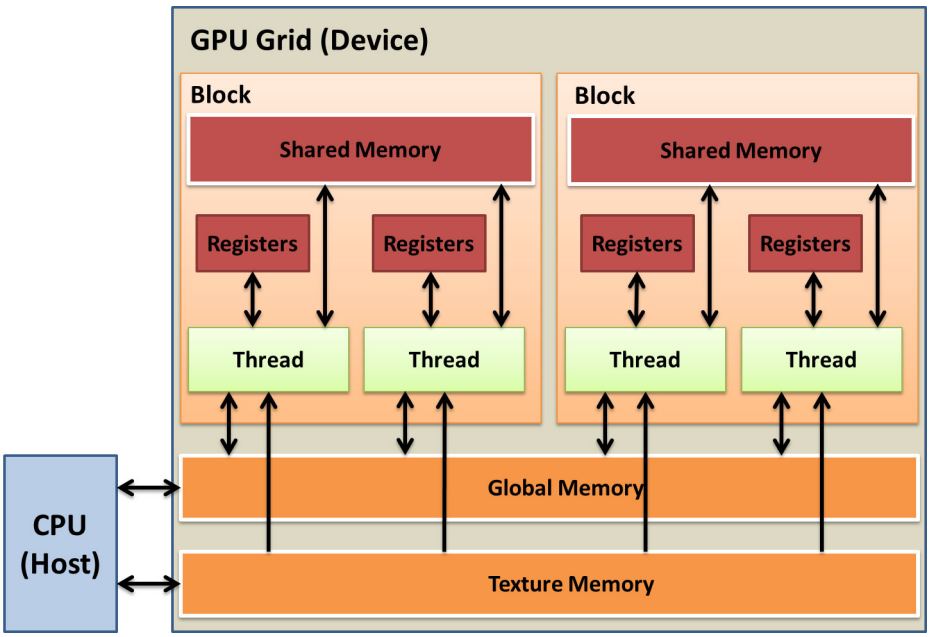

Figure I.9: The GPU memory architecture (the diagram is credited to yuwang-cg.com (2014)).

In most computer systems GPUs function as co-processing units sitting on graphics cards. All information is on some storage medium (e.g. HDD or SSD) on the CPU (the host) side and data is loaded from storage mediums into the operative memory of the CPU (the host memory). The GPU (device) communicates with the CPU over a PCI express bus (∼1-16GB/sec). In host-device communication data is transferred from host memory to the memory of the graphics card (global memory or GMEM and constant memory or CMEM).

GMEM is the main - the largest and slowest - memory space of the device. CMEM is a limited-size (64kB) with broadcast capability and with its own special cache. Beyond the globally shared memories, GPUs also have a number of registers and some limited shared memory (SMEM) to work with. SMEM is faster than the GMEM and also allows parallel access but is only visible to the CUDA block it belongs to (see Figure I.9). SMEM

is the medium that allows intra-block communication between threads. Between different blocks communication can only happen across the GMEM. The last memory type available is texture memory. The use of this memory has special requirements and is less useful for general-purpose HPC.

The graphics card can hold multiple streaming multiprocessors (SMs), each of which contain a large number of GPU cores. All SMs and GPU cores share the CMEM and GMEM. Much like CPUs, GPUs also have a multi-level cache system that isn't directly accessible, but allows optimisations with the right memory-access strategies.

Table I.1: GPU memory types

Mem. type	Size	Bandwidth	Latency	Further notes
GMEM	1-12GB	∼32-320GB/s or cache speed	200-800 cycles on a cache miss	Observed performance can be greatly improved due to L1 and L2 cache and switching between warps
L2 cache	1.5MB shared across SMs	∼500GB/s	200-300 cycles	Works with granular (32-128B) cache lines, largely hidden from the programmer
L1 cache	16-48kB	∼500-1600GB/s	10-80 cycles	Works with granular (32-128B) cache lines
CMEM	64kB	4b/cycle or cache speed	20-200 cycles dependent on cache misses	Optimised for broadcast and access is accelerated by a special 8kB dedicated constant cache
SMEM	16-48kB per block	up to ∼2.5TB/s	20-30 cycles	Allows parallel access (16-32 banks), not cached
Registers	32 bits	∼8TB/s	-	21-255 registers per thread (depends on the architecture and the number of threads used)

These figures are characteristics of the current NVIDIA Fermi and Kepler architectures and are expected to improve in future architectures.

GPUs are able to run 4-12 blocks on each SM, with 4 warps running in each block. The result is about 32-48 warps and 1024-1536 threads running in the same time per SM. With multiple SMs (2-15), the maximum concurrent execution achievable is about 2-23k.

GPUs have a few further advantages over CPUs. Due to the relative independence of threads, GPU scheduling may be able to hide the latency of memory access by switching between warps (the 32-48 warps on each SM). This means that memory access latencies may be hidden (completely with 10+ operations per memory access) and programmers usually benefit from

a large number of blocks and threads in ways that is unique to GPUs.

Since memory operations often create a performance bottleneck, GPU code largely depends on the optimal use of the different memory types (Table I.1) using coalesced mapping and coalesced access. Coalesced memory access means that threads that are expected to execute in the same warp access consecutive parts of the memory. This allows both for optimised caching between GMEM and the register as well as parallel access to SMEM. It doesn't hurt if CMEM is also accessed in a coalesced way.

3.1.3 Parallel programming under CUDA

Besides the CUDA programming model and the differences of the available memory types, there are a fair number of particular technicalities one needs to be aware of when programming under CUDA. Below is a high-level explanation of some key details that are particularly relevant to this work (mainly to chapter III but also to chapter IV).

Kernels and launching kernels

The core code units in CUDA are kernels. Kernels are pieces of code that are given to the device to execute concurrently. The sequential program of a kernel is executed by every thread.

The launch of kernels is preceded by host-device memory transfers and are usually followed by device-host memory transfers. Setting up a kernel run after launch may also involve parallel data-loads into shared memory (meaning GMEM to SMEM transfers). The kernel launch itself has some minor overhead, but the SMEM loads may be more significant.

Threads, blocks and synchronisation

Threads can access their thread and block id's for identifying relevant pieces of data to work on (this is mapping in map-reduce terminology). Often threads need to synchronize when there is inter-thread data dependency.

Due to the CUDA programming model, blocks can't ever synchronise during a kernel run, block-synchronisation can only be achieved by the host between kernel runs and should hence be avoided if possible.

Parallel reduction

Most parallel algorithms require reduction operations (e.g. large-scale summations, finding minima, maxima, etc). These operations can not be fully parallelised - the best performance can be achieved by tree-based semi-parallel summation algorithms that take $\log(\#threads)$ steps (see Figure I.10).

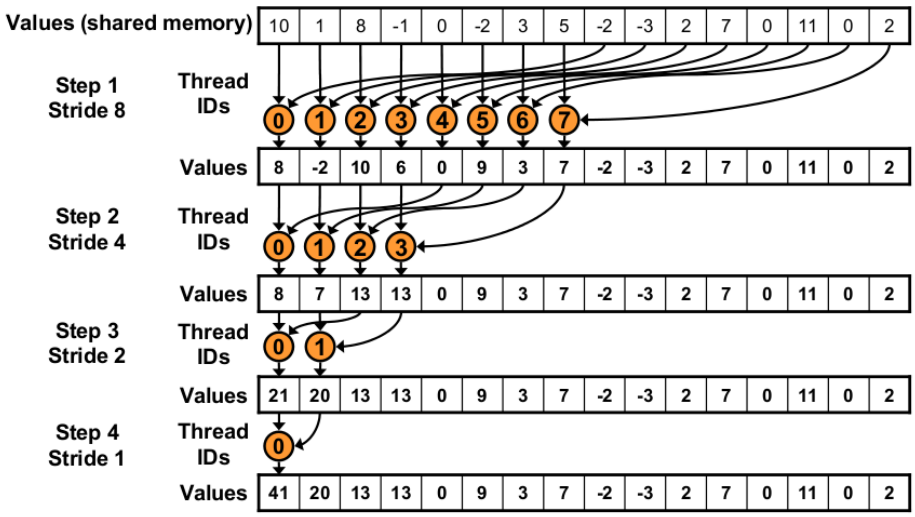

Figure I.10: A parallel reduction example - parallel summation (NVIDIA).

Paging virtual threads

In case there is a need for more threads than optimally advisable (or available) for an application, it is possible to use a large number of 'virtual threads' over a smaller number of actual CUDA threads. Virtual threads can be divided into bundles of threads. The bundles share the actual CUDA threads sequentially. Calculation can be performed only on one bundle at a time, but bundles can be cycled through, swapping or 'paging' them in and out, similarly to how virtual memory works in modern operating systems. Paging works by saving and loading key thread variables.

This method introduces some forced sequentiality in implementations, because computation on different bundles has to be performed one after the other. The communication between bundles also introduces extra memory operations, usually over GMEM. However, virtual threads also allow the use of an optimal number of threads. For the use of virtual threads in HMM algorithm implementations see (Cartey et al., 2012).

Streaming

The necessity of streaming comes from the limited amount of memory available in GPUs. It is often necessary to divide computation into parts that are then solved in parallel. Naturally, kernel 'input' data needs to be transfered to the device and results need to be transferred back to the host. Streaming is a method available in CUDA that allows this procedure to be performed efficiently. In a simplistic scenario, while calculation is performed on part i, the results of part $i-1$ are being transferred to the host and the input for part $i+1$ is being transferred to the device. This setup allows the two-way transfers, GPU calculations and CPU calculations to be overlapped.

Streaming requires host memory to be pinned, which means disallowing it to be paged by the operating system. Pinning may greatly reduce the (system-level) performance of the host when the pinned memory is large.

4 Parallel algorithms for HMMs

In statistics, and especially in statistical genetics, it is essential to have sufficiently fast algorithms for inference that can cope with large amounts of data in an acceptably short time frame. The traditional way of achieving higher speeds is through simplifying models and optimising algorithms. Although the standard approach of designing efficient algorithms should always be the first to be considered in development, it is usually applicable only within limits - over-simplifications can often lead to decreased statistical performance.

With the advent of affordable parallel computing brought by GPUs, it is possible to achieve faster software on a wide scale through the development of parallel algorithms. Parallel computing can open doors to the applicability of methods previously seen as being inapplicable purely due to long run times. Unfortunately, making use of parallel computing requires a change in the way that algorithms are developed, a change from 'sequential thinking' to 'parallel thinking', but the parallel approach could revolutionise the way we see certain methods and statistical problems in general (Lee et al., 2009; Nickolls and Dally, 2010).

An area where the power of parallel computing and specifically the capabilities of GPUs may be successfully utilised is the improvement of HMM algorithms. A particularly significant set of applications in genetics are based on the widely used LS model (Li and Stephens, 2003) which are particularly suitable for parallelisation over GPUs.

4.1 Approaches and previous works

Recently there have been some advancements in both the theory and practice of parallelising HMM algorithms using various different approaches that seem to have resulted in widely varying results in execution time improvements (Cartey et al., 2012; Du et al., 2010; Horn et al., 2005; Hymel, 2011; Li et al., 2009; Liu, 2009; Nielsen and Sand, 2011; Sand et al., 2010; Turin, 1998; Zhang et al., 2009).

Essentially, the core process in all HMM algorithms is filling out a dynamic programming table, which is performed very similarly across these algorithms (see Figure I.2). The two most trivial approaches for parallelisation are a) over multiple observation sequences, and b) over multiple models or parameterisations. In these two cases the algorithms are run in parallel but completely separately for each observation, model or parameterisation. These two approaches can be easily implemented on GPU-based, CPU-based and even distributed systems and will be referred to as the 'trivial' parallelisation approaches in this thesis.

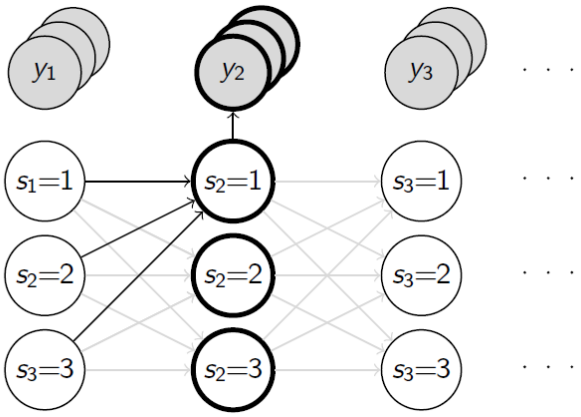

Figure I.11: Parallelisation of HMM algorithms over the observation set and over the state space (the 'present' states). Computation for the bold circles may be performed in parallel.

The next level of complexity in HMM parallelisation is when calculations are performed parallel for different states of the HMMs. In most recursive HMM algorithms some value needs to be calculated for each state s_l at each step of the recursion. The corresponding operations can be performed in parallel for each state s_l. When implemented, this approach requires a high degree of communication (i.e. data sharing) between parallel threads, which makes GPUs the most suitable parallel architecture for this task at present. Parallelisation over the state-space will be referred to as the 'basic' approach or 'state-parallelisation' in this thesis. Most published parallel HMM methods make use of the trivial and the basic

approaches (see Figure I.11).

An alternative parallelisation approach is one where computation is run in parallel over the length of the sequence. (Turin, 1998) pointed out the theoretical possibility for such an approach. A practical treatment and an implementation for small state spaces is given in (Nielsen and Sand, 2011). This third approach is going to be referred to as the 'block-parallel' approach and is further discussed and developed in chapter IV.

Chapter II

GENOME-WIDE SINGLE-LOCUS VARIANT CALLING USING GENOTYPE LIKELIHOODS AND HAPLOTYPE SCAFFOLDS

1 Introduction

This chapter presents and discusses a novel single-locus variant caller method (SLVC) for making phased haplotype calls based on high-uncertainty genotype likelihood (GTL) data. SLVC also assumes the availability of phased low-uncertainty marker (e.g. SNP) data for the same individuals. The low-uncertainty marker data is used as a scaffold when performing inference on every locus independently and is key to guide inference by linking information between individuals. The scaffold is assumed to provide sufficient context information, allowing for the single-locus method to retain high accuracy compared to multi-locus methods, while achieving better computational performance and possibly higher consistency due to its reduced search space. Besides phased haplotype calls, SLVC also provides posterior probabilities for use in downstream analysis.

SLVC was inspired by the IMPUTE2 method (Howie et al., 2011, 2012; Howie and Marchini, 2010; Howie et al., 2009; Marchini et al., 2007) and might be thought of as a simplified, single-locus version of the haplotype estimation (phasing) part of IMPUTE2.

The original intention with the SLVC method was to produce high-quality reference indel data for downstream analysis and thus integrate indels (or possibly other structural variants) into the workflow of genome-wide association studies (GWAS). Consequently, most evaluation experiments were performed with a focus on indels. SLVC has been proven to be applicable to indels, but is also applicable to other types of genetic variation in its current form. While the enhancement of indel-specific variant calling has not been achieved as intended, the method offers a fast, accurate and extremely highly parallelisable alternative to other HMM-based phasing and calling algorithms.

The development of SLVC was concluded in 2012 when the method SHAPEIT (Delaneau et al., 2012, 2013a,b; O'Connell et al., 2014) was released. SHAPEIT was expected and has since been proven to be a fast, parallelisable and an exceptionally accurate method for phasing genetic variants. Recently, SHAPEIT has been used extensively in the 1000 Genomes Project (1000GP) for producing high-quality phased reference data on a large scale (Delaneau et al., 2014).

2 Data

The SLVC method was designed to use GTL data as its input. Method development was carried out in light of the properties of a prototype indel GTL dataset produced by the method Dindel (Albers et al., 2010) under the 1000GP. Analysis of this dataset is presented in Section 2.1.

The secondary input, phased SNP haplotype data, was provided by Phase III of the HapMap Project (The International HapMap Consortium, 2007, 2011) and indel validation data was produced by Sequenom under the 1000GP pilot. The validation data contained low-uncertainty unphased genotype calls for a number of indels of HapMap individuals. With no further alternatives available for validation, some evaluation tests were performed using semi-synthetic pseudo-indel data in order to assess the accuracy of phased HT predictions.

2.1 Indel data from the 1000 Genomes Project

This section introduces the indel GTL data used during development and presents preliminary analyses exploring the LD structure of indels.

Read data from shotgun sequencing

The technology that enabled the production of GTLs by Dindel is called 'shotgun sequencing' (such as Illumina Solexa, ABI SOLID and 454). It is common to these technologies that sequencing is done in two stages. First, a large number of short reads, called 'shotgun reads', are produced experimentally at random points of the genome. Next, the short (and mostly overlapping) reads are assembled to produce complete sequence information. The assembly process is built on the fact that the genomes of any two humans differ by less than 0.1%. Consequently, the position of short reads may be mapped by aligning them with a reference sequence despite the possible presence of variation. Further, when a larger number of individuals are sequenced at the same time (or some reference genotype information is available), information may be pooled between sequences in order to fill in gaps (The 1000 Genomes Project Consortium, 2010).

A key advantage of next-generation sequencing techniques is that they are very cost and time effective, which is particularly important when sequencing the whole genome (about three billion base pairs) for a large number of individuals. Next-generation sequencing allows for studying genetic variation on a fine scale with large samples, such as in the 1000GP. The high-quality sequence data on populations of individuals that these methods provide uniquely enables the capture of structural variants such as indels.

Indel genotype likelihood data from Dindel

The protoype GTL dataset used for development was provided by Dindel (Albers et al., 2010), a method for calling indels from next-generation sequence read data. It relies on external methods to map reads to the reference and to provide a list of candidate indels and 'mapping quality' measures. Both the sensitivity of finding indels and the accuracy of Dindel depend heavily on the read mapper, and so the best results are obtained by using mappers that are optimised for indels, such as Stampy (Lunter and Goodson, 2010). Dindel produces indel calls with a low false-positive rate using a probabilistic realignment method (see Figure II.1) with the ability to deal with base-calling and mapping errors, as well as higher expected sequencing errors in long homopolymer runs.

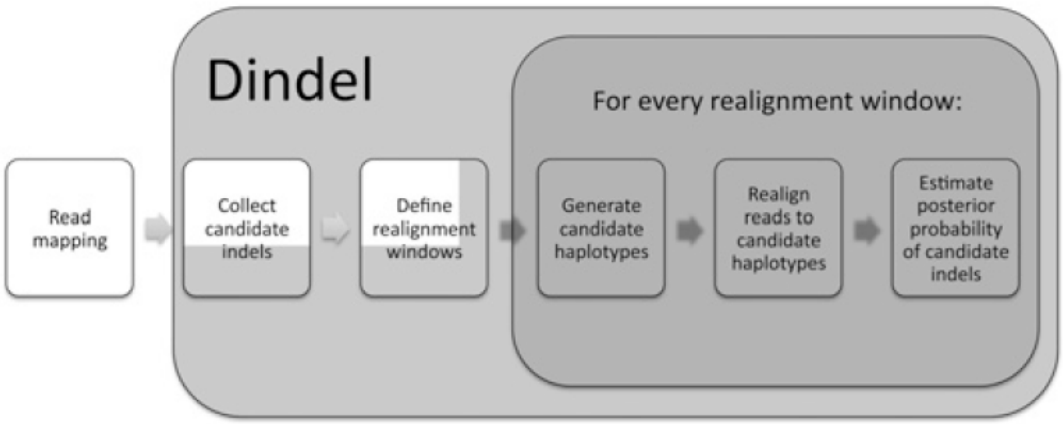

Figure II.1: Outline of the Dindel algorithm (Albers et al., 2010).

Dindel provides information on the location of indels, their type (insertion/deletion), the exact details of variation (the inserted/deleted basepair sequences) and the GTLs for each possible genotype (GT): homozygous reference, heterozygous and homozygous variant genotypes (coded as genotypes 0, 1 and 2, respectively).

In most cases, it is posterior GT probabilities (GTPs) that are actually used in SLVC. These GTPs are calculated from GT likelihoods simply by normalization, assuming a uniform prior.

2.1.1 Reliability and accuracy

The first step was to evaluate the reliability of Dindel GTLs (and corresponding GTPs). This assessment was performed by comparing the GTP modes with the validation (Sequenom) GT calls. The Sequenom data and the GT likelihood data agreed in both position and variation simultaneously in only 96 indels of 59 individuals of the CEU subset. Consequently, the comparison of datasets was performed on this intersection.

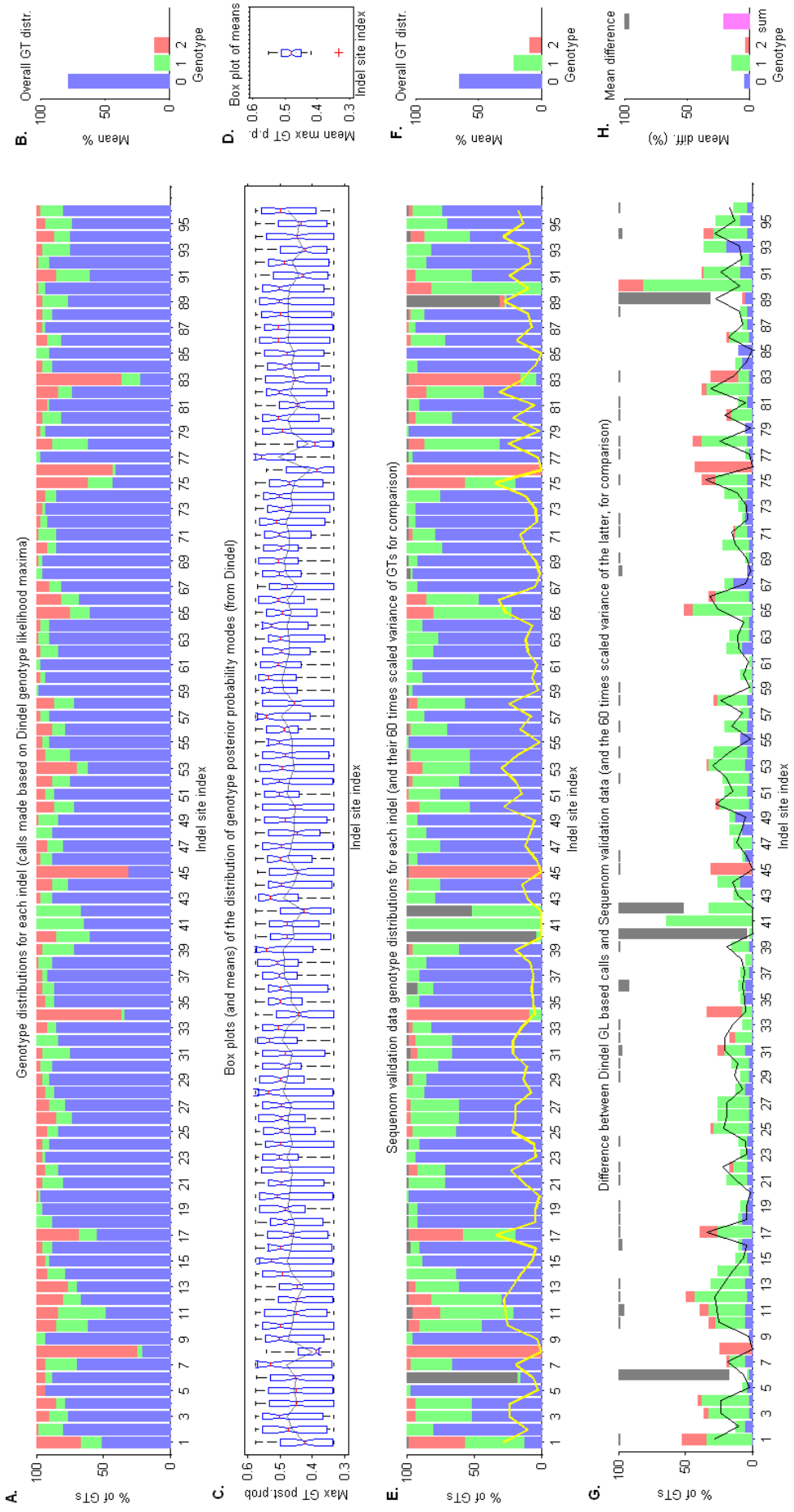

Figure II.2: Comparison of Dindel GTL modes and Sequenom calls. The top row shows (A) the distribution of Dindel GTL modes for each site, and (B) their means per GT. The second row presents the uncertainty of the Dindel GT calls as (C) a box plot of the distribution of GT posterior probability modes for each site, and (D) the distribution of their means across individuals. The grey line shows the mean of GTP maxima for each site. The 3rd row shows (E) the distribution of Sequenom GT calls for each site, and (F) their means. The yellow line shows the variance of GTs per site. The 4th row shows (G) the difference between Dindel GTL modes and Sequenom calls per site per GT and (H) their means. The black line on the left is again the GT variance in the Sequenom data. The colouring of bar graphs refers to GTs as follows: blue - homozygous reference, green - heterozygous and red - homozygous variant. Grey bars show the percentage of missing data in the Sequenom set.

There were only 96 indels agreeing (in both position and variation) between the Dindel and Sequenom sets on chromosome 20 for 59 CEU HapMap individuals. The Dindel and Sequenom datasets are compared in Figure II.2. From the comparison of Figures II.2 A with II.2 E and II.2 B with II.2 F it is clear that the distribution of GTs is considerably different between the Dindel modes and the Sequenom calls. The actual percentages of GT disagreement between the two sets are shown for each site in Figure II.2 G and II.2 H. This disagreement seems to be varied and fluctuating over sites but their majority fall into the category where GTs are called heterozygous (GT=1) in the Sequenom data. This type of disagreement makes up about 70% of the presumed errors in the Dindel dataset, as an overall 14.7% of all 5439 GTs have their highest likelihood at 0 or 2, while they are simultaneously called heterozygous in the Sequenom data.

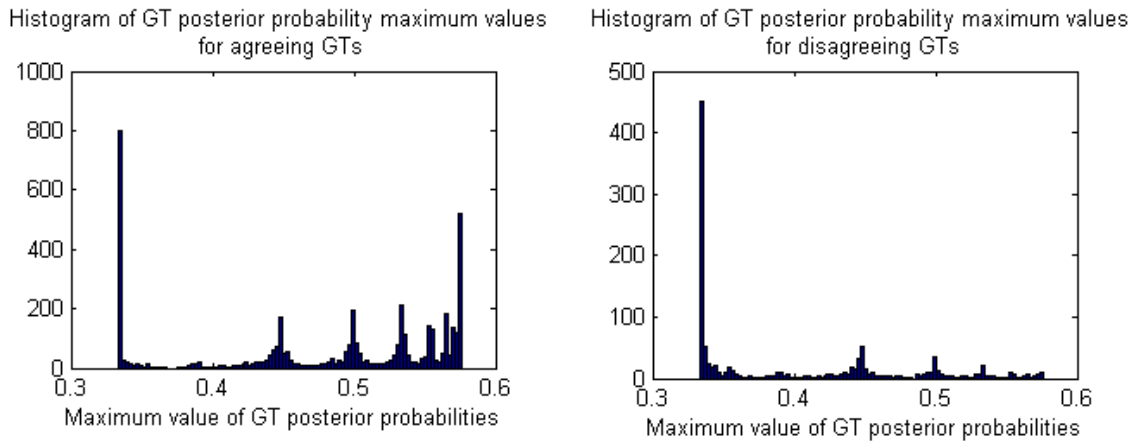

Figure II.3: Comparing the distribution of GTP maxima of GTs for which the Dindel and Sequenom data agree, against GTP maxima for GTs for which the datasets do not agree and the Dindel GTPs are thought to be misleading.

It is reasonable to assume that GT calls based on GTPs are less reliable when the given GTP triplets are more uniform (flat). The uniformity of GTPs is well represented by the distribution of GTP maxima. The box plots of Figures II.2 C and II.2 D do not seem to show a clear relationship between the maximum of GT posterior probabilities and differences between the Dindel modes and the Sequenom data, but further examination reveals a strong dependence. Figures II.3 A and II.3 B and show that there are both agreements and disagreements in the case of completely flat GTPs (as expected) but higher GTPs tend to correspond to the correct GTs. This may be either because the Dindel likelihoods are reasonably calibrated or because in the case of flat likelihoods, the maximum GT probability value is not much higher than the rest and, hence, correctness of GTL mode based calls are more random.

The last validation analysis performed on the two datasets characterised the disagreement

Table II.1: Confusion matrix comparing GT calls from the Sequenom validation data and the GTL modes of Dindel. The 4^{th} column and row contain row and column sums, respectively. The 5th row shows the number of missing genotypes in the validation data by the genotypes given by Dindel, and row 6 presents the total number each GT is called based on Dindel GTL modes.

		Dindel GTs (modes)			
		Hom.ref.(0)	Het.(1)	Hom.var.(2)	Sum
Sequenom GTs	Hom. reference (0)	3494 (64.2%)	101 (1.9%)	86 (1.6%)	3681 (67.7%)
	Heterozygous (1)	618 (11.4%)	450 (8.3%)	180 (3.3%)	1248 (22.9%)
	Hom. variant (2)	147 (2.7%)	27 (0.5%)	336 (6.2%)	510 (9.4%)
	Sums	*4259 (78.3%)*	*578 (10.6%)*	*602 (11.1%)*	*5439 (100%)*
	Untyped/missing	*164*	*41*	*20*	*225*
	Sums incl. untyped	*4423*	*619*	*622*	*5664*

between them. The disagreements are characterised by a confusion matrix (Table II.1). The percentage figures show that the majority (11.4% discordance out of the total 21.3%) of all disagreements fall into the type when Dindel $\arg\max(GTP)$=0, while the corresponding GT call in the Sequenom data indicates it to be heterozygous (GT=1). The second most frequent type of disagreement (with a 3.3% contribution) relates to cases when Dindel modes favour homozygous variant GTs where the Sequenom calls are 1. In general, Dindel seems to be under-predicting heterozygous GTs and makes too many homozygous reference predictions. The number of heterozygous Sequenom GTs marked to be heterozygous by Dindel GTL modes (450 cases) is actually lower than those marked incorrectly (798), which highlights some complications when using the Dindel dataset.

Overall, when working with the Dindel data, the GT likelihoods may be uninformative or misleading when used for calling directly by themselves. Moreover, one can expect that the likelihoods for the heterozygous sites are the most unreliable for making calls from directly.

The disagreement between the Sequenom and Dindel sets is expected, since the two data sets are produced with completely different technologies. The Dindel data is based on low-coverage short reads, which directly implies a higher degree of uncertainty and missing data that Dindel has to work with when producing the GTLs. It is natural that this lack and uncertainty of experimental data leads to relatively flat GTLs and potentially a higher degree of errors. While the Sequenom set is expected to be more reliable, we can expect some sequencing errors there too (the extent of which is not explored here).

It is important to emphasize that all the accuracy analyses assumed that the Sequenom data

is 100% correct. Since that is unlikely to be the case, all the above presented figures should be taken as approximate results.

2.1.2 Exploring LD

An important step when analysing the prototype indel data was to explore the co-occurrence of SNPs and indels and to compare the LD structures between SNPs and indels. The analysis was done by calculating R^2 between the GTs at each site in 60 CEU HapMap individuals. The SNP GTs used came from the 1000 Gnomes project SNP haplotypes of the same individuals and the indel GTs were called according to Dindel GTL modes.

Observing Figure II.4, it appears that LD between indels (Figure II.4 B) as well as indels and SNPs (Figure II.4 C) follows 'patchy' patterns with block structures very similar to those observed between SNPs (Figure II.4 A). LD is generally higher between neighbouring loci than between those far apart, as expected. It seems, however that the locations of high LD blocks between indels do not fully agree with the high LD patches between SNPs, while the patterns of high LD between SNPs and indels better match to the LD patterns between SNPs.

The differences between Figures II.4 B, E and II.4 A, D suggest that LD between indels is, interestingly, of a slightly different nature. LD between close indels is relatively lower than LD between close SNPs but LD between distant indels can be relatively higher than LD between distant SNPs. Hence, it may be assumed that LD between indels is less distance dependent and the LD structure between indels may be slightly more complex than between SNPs.

While the LD pattern between indels and SNPs generally agrees with that observed between SNPs, the local LD (LD in 4kb wide windows) is relatively lower (see Figures II.4 C,F) and distant LD is of similar strength with a similar trend of decay. The LD patterns and LD decay shown in Figure II.4 are typical throughout the non-centromere parts of chromosome 20.

It may be useful to note that the number of indels in any region is lower than the number of SNPs, which makes the resolution of the LD plots somewhat unbalanced. A limitation of the LD assessment is that given the extent of unreliability of GTL mode-based indel calls (Section 2.1.1), it can be assumed that there are a significant number of errors in these analyses. Nevertheless, the results gave sufficient indication that extracting information using LD between SNPs and indels may be a viable direction.

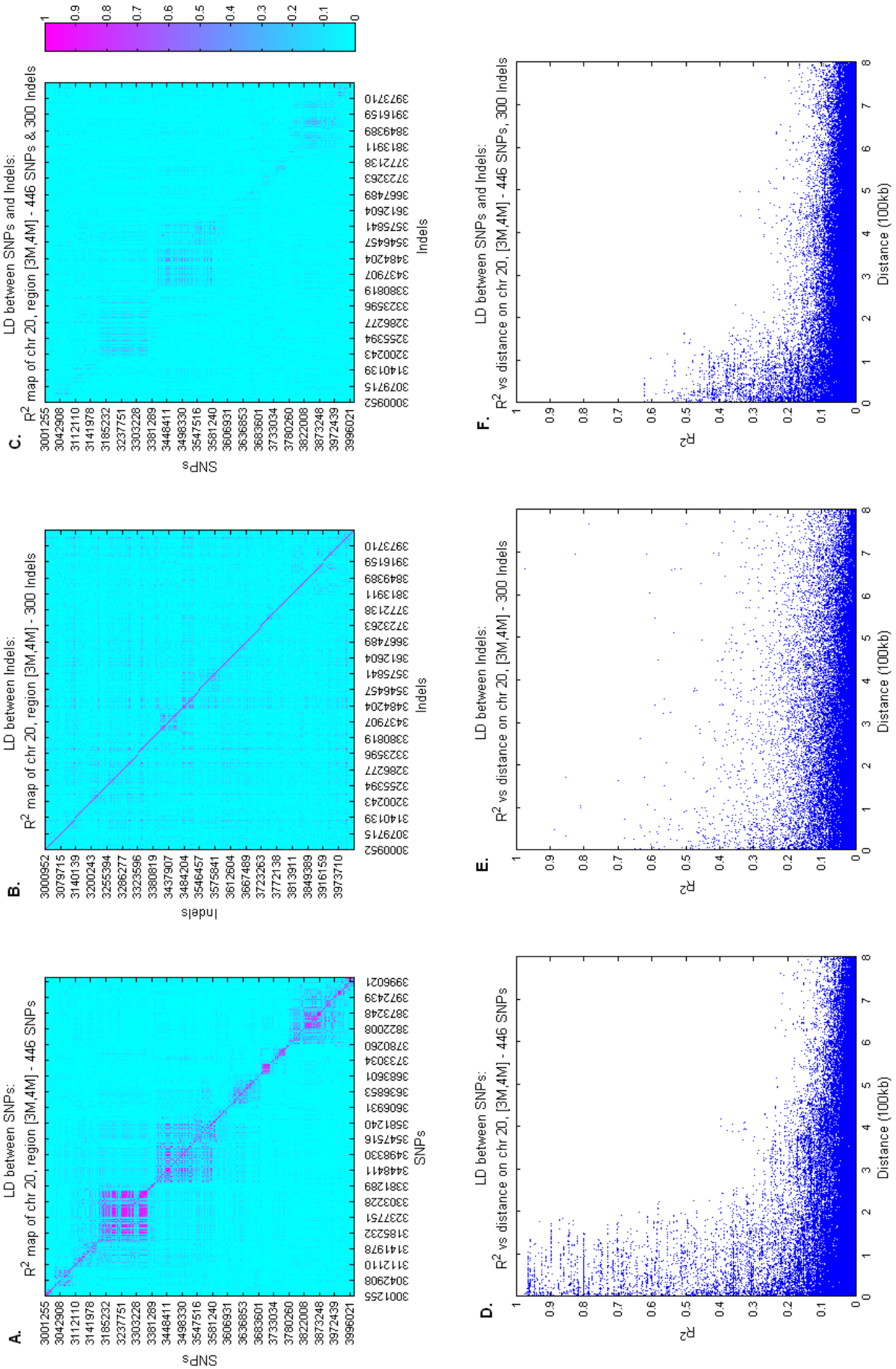

Figure II.4: LD between SNPs and Indels. The heat plots of the top row show the correlation (R^2) - (A) between SNPs only, (B) between indels only and (C) between SNPs and indels. All variants are in the same 3Mb-4Mb region of chromosome 20 of the same 60 HapMap individuals with European ancestry (CEU). The second row includes corresponding plots of LD decay with physical distance (R^2).

2.2 Pseudo-indel data

In the absence of phased indel validation data, a semi-synthetic pseudo-indel dataset was produced and used for evaluating the accuracy of SLVC in terms of phased HT predictions. This pseudo-indel dataset was produced by blending real SNP data with the characteristics of the Dindel dataset. The aim with the pseudo-indel dataset was to have phased HT validation data and corresponding GTP data that have the same characteristics as a real indel data were assumed to have.

Pseudo-indel synthesis started by randomly selecting a number of SNPs from a phased haplotype panel. The chosen SNPs were marked as validation data and were also used as the basis for synthesising high-uncertainty pseudo-indel GTPs according to the characteristics of the Dindel set. Using real SNPs as a basis for pseudo-indel synthesis ensured that the LD structure between pseudo-indels was realistic, since LD between SNPs was assumed to be very similar to LD between SNPs and indels (according to results in Section 2.1.2).

In order to simulate the characteristics of a real indel dataset, the selection process for SNPs (to be pseudo-indels) was designed to ensure that the overall frequency of homozygous reference, heterozygous and homozygous variant genotypes matched the corresponding frequencies observed for indels in the Sequenom dataset ($f(0) = 0.69$, $f(1) = 0.21$, $f(2) = 0.094$). Next, an 'error type' (confusion matrix entry) was chosen stochastically for each pseudo-indel by sampling Dindel's confusion matrix (Table II.1). Next, a GTP triplet was sampled from Dindel's GTP distribution according to the chosen error type. The GTP distributions are shown in Figure II.5 corresponding to each confusion matrix entry.

For the evaluation experiments $N = 84$ CEU individuals were chosen from the 1000 Genomes sample and 99 SNPs were chosen to be pseudo-indels from the 76Mb-81Mb region of human chromosome 15.

3 Methods

This section presents a method, SLVC, for finding the phased HTs for a whole sample of individuals based on given GTL data and the make-up of flanking SNP haplotypes. Although the SLVC method works equally well with SNPs and indels, the method will be presented assuming that it is used for indel calling.

The flanking SNPs are used to provide a fixed haplotype scaffold and to provide information on the genetic history of individuals that make up the sample, which then guides the inference process; Figure II.6 illustrates this setup. The GTP triplet bars highlight some of the typical

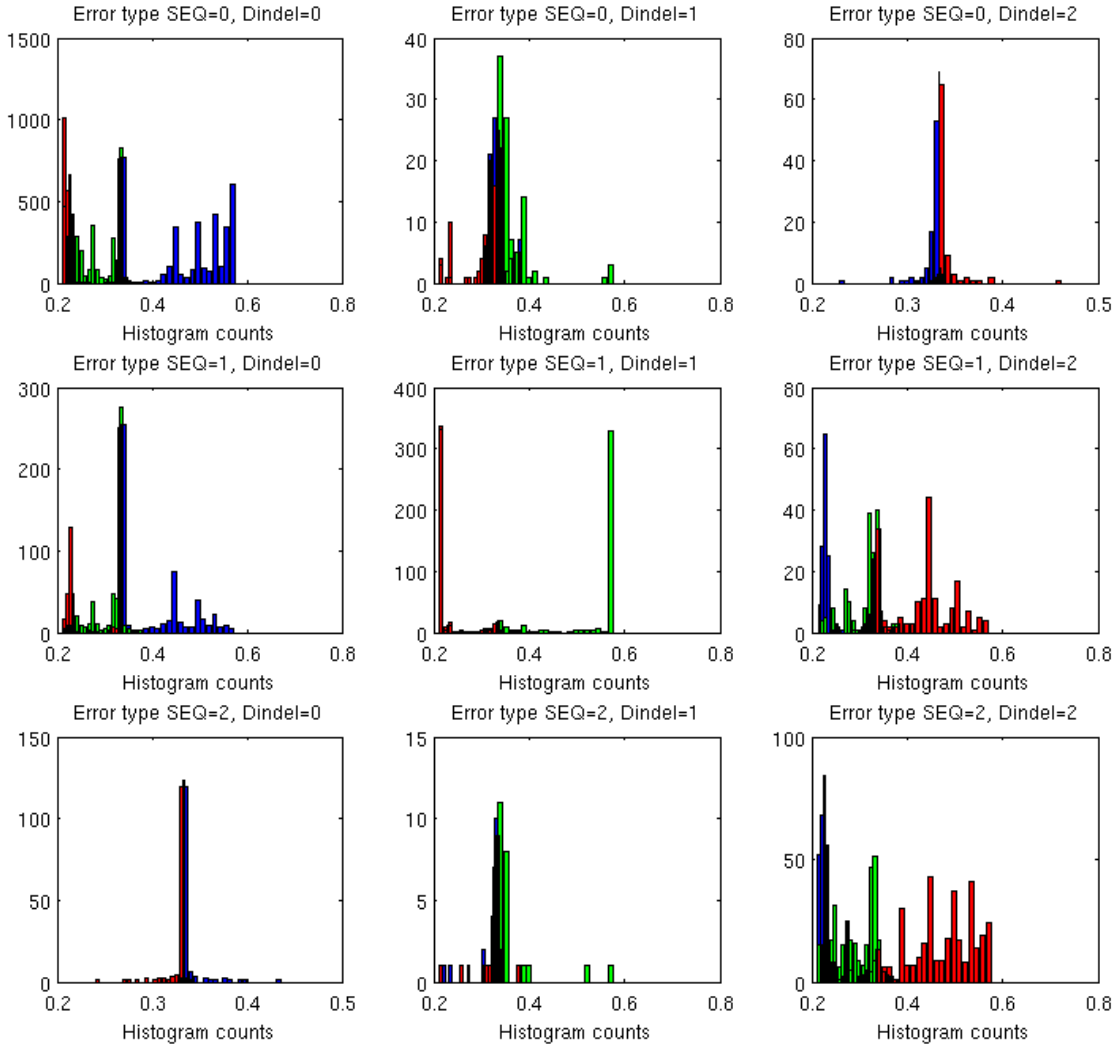

Figure II.5: The GT probabilities provided by the Dindel dataset are arranged into a confusion matrix according to the cell they correspond to. Each row corresponds to a true GT (with the Sequenom calls taken as the truth). The GTP triplets provided by Dindel are classified into columns according to their mode. The resulting figure resembles a confusion matrix with a representation of the GTP distributions corresponding to the 3 matching cases (in the main diagonal) and the 6 error types.

scenarios and challenges, such as having flat GTL triplets (individual D) or high likelihoods for GTs 0 and 2, and low likelihood for GT 1 (Individual C). Individual E demonstrates the case where the task is determining the chromosome label of a pair of 0 and 1 alleles when the GT is expected to be 1 (Individual E).

3.1 Model

The core of the SLVC method is a single-locus model of unobserved alleles which links sequencing read information and scaffold haplotype information. Before describing the model,

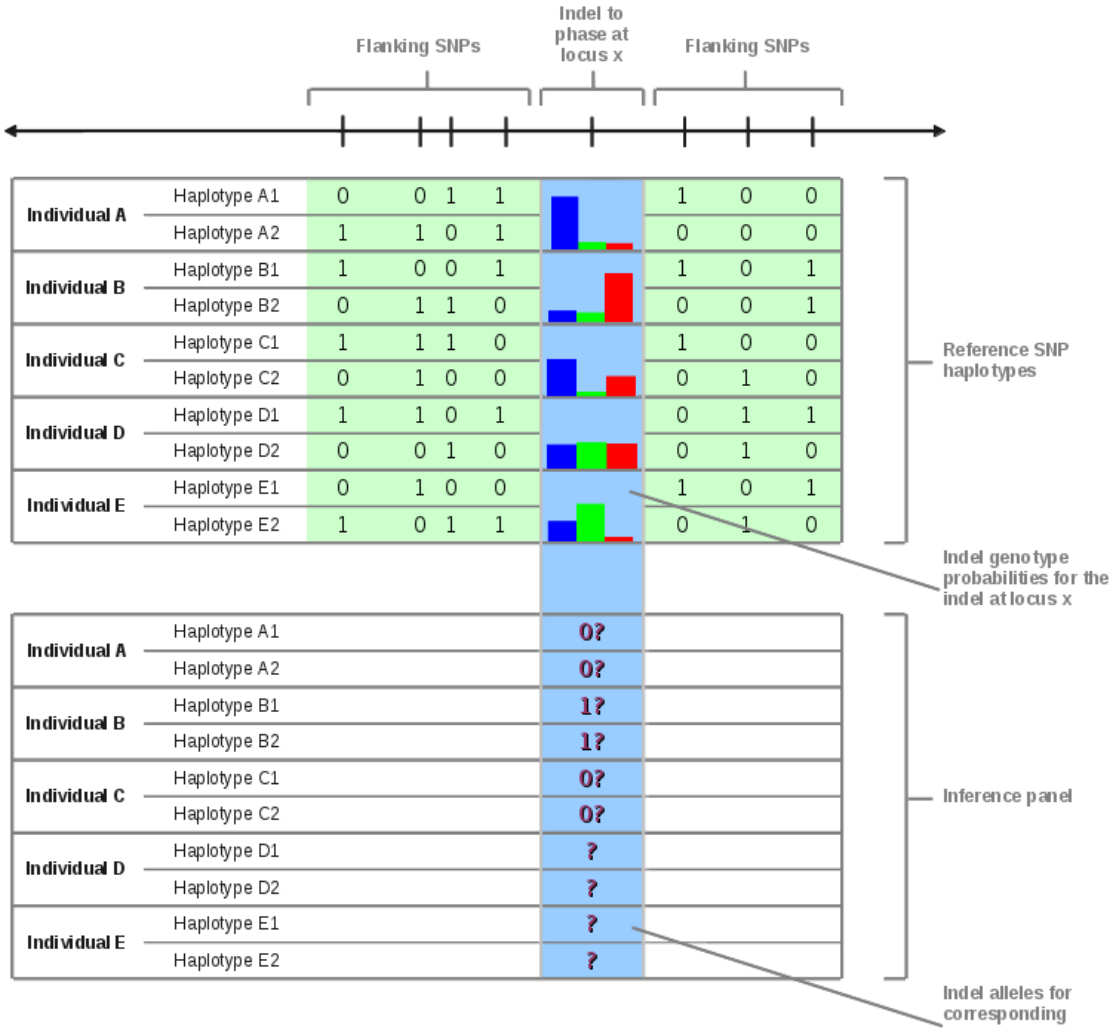

Figure II.6: The indel phasing problem: phasing indels into given SNP haplotypes based on Indel GT probabilities for the same individuals.

the following notation is established:

Notation

- x is the base pair position of the putative variant we are interested in.
- N is the number of individuals in our sample, n is the running index of individuals.
- R is all sequencing reads. Although the read data is not directly observed in SVCL, only through GTLs, it will be used to explain our model.
- R_n refers to the sequencing reads at the site of interest that are summarised in the GTLs of individual n.
- $GTL_n(g) = \log P(R_n|g)$ is the indel GTL i.e. the log-likelihood for genotype g of

individual l. The genotype g could be 1 of 3 possibilities: {0 = homozygous reference, 1 = heterozygous, 2 = homozygous variant}.

- Let $H = \{H_1, ..., H_{2N}\}$ be the $2N$ known SNP haplotypes of the N individuals.
- Let $Z^x = \{z_1, ..., z_{2N}\}$ be the vector of (unobserved) indel alleles that we are looking for at position x, i.e Z_i^x will be the indel allele in the i^{th} haplotype at position x.
- H^* is the combined set of haplotype data H and the indel configuration Z^x of the putative variant inserted into these haplotypes at the appropriate point.

3.1.1 Model derivation

To phase individual indels, we are interested in the posterior distribution of unobserved indel alleles (Z^x) conditional on the read data R and the flanking haplotype data H

$$P(Z^x|H, R) \propto P(R|Z^x, H)P(Z^x|H). \tag{II.1}$$

The first term of the right-hand side is the probability of the read data given a configuration Z^x. Treating Dindel as a blackbox method for simplicity, we assume that the reads R are conditionally independent of the flanking SNP haplotypes H given the configuration Z^x, hence $P(R|Z^x, H) = P(R|Z^x)$. This term can then be written in terms of the genotype likelihoods as the product

$$P(R|Z^x) = \prod_{l=1}^{N} P(\ R_l \mid g = \{Z^x_{2(l-1)+1}, Z^x_{2l}\}\ ). \tag{II.2}$$

over the indel GTL triplets of each individuals in our sample. The second term is crucial, as it relates the indel configuration Z^x to the flanking SNP haplotypes. It can be thought of as a prior on the different configurations of the indel.

The first attempt to approximate the likelihood was to use a 'pseudo-likelihood' (Besag, 1974) in which the probability of the indel configuration Z^x was written as the product of probabilities of each element of Z_i^x, given all the haplotypes H and all the other elements which were denoted Z^x_{-i}.

$$P(Z^x = z|H) \approx \prod_{i=1}^{2N} P(Z_i^x = z_i|H, Z^x_{-i}) \tag{II.3}$$

Unfortunately, approximating the likelihood this way did not seem to provide sufficiently accurate results (see Section 3.2.2). Consequently, development proceeded with inference methods that did not require a closed form of the posterior, only the conditional distributions

$P(Z^x_{k_1}|H, R, Z^x_{-k_1})$ and $P(Z^x_{k_2}|H, R, Z^x_{-k_2})$ in order to sample from the posterior (such as Gibbs sampling). These conditional distributions can be expressed by writing

$$P(Z^x_i = z_i|H, R, Z^x_{-i}) \propto P(R|Z^x_i, Z^x_{-i}, H)P(Z^x_i|H, Z^x_{-i}), \tag{II.4}$$

where

$$P(Z^x_i = z_i|H, Z^x_{-i}) \propto P(Z^x_i = z_i, H_i|H_{-i}, Z^x_{-i}) = P(H^*_i|H^*_{-i}), \tag{II.5}$$

which has the form of the probability of a haplotype H^*_i given a set of other haplotypes H^*_{-i}. For this term we can use the HMM based Li and Stephens model (Li and Stephens, 2003), in which the haplotype H^*_i is modeled as an imperfect mosaic of the other haplotypes H^*_{-i}. Using this model, if we use S to denote the underlying hidden state vector of the HMM, then we can write

$$P(H^*_i|H^*_{-i}) = \sum_S P(H^*_i, S|H^*_{-i}). \tag{II.6}$$

If we denote the underlying state at position x of haplotype i as S^x_i, then we can also write

$$P(Z^x_i = z_i|H, Z^x_{-i}) \propto P(H^*_i|H^*_{-i}) = \sum_{S_{xi}} P(H^*_i, S^x_i|H^*_{-i}), \tag{II.7}$$

which can be simplified as

$$\begin{aligned} P(Z^x_i = z_i|H, Z^x_{-i}) \propto P(H^*_i|H^*_{-i}) &= \sum_{S_{xi}} P(H^*_i, S^x_i|H^*_{-i}) \\ &= \sum_{S^x_i} P(Z^x_i = z_i, H_i, S^x_i|H_{-i}, Z^x_{-i}) \\ &= \sum_{S^x_i} P(Z^x_i = z_i|H_i, S^x_i, H_{-i}, Z^x_{-i})P(H_i, S^x_i|H_{-i}, Z^x_{-i}) \\ &\propto \sum_{S^x_i} P(Z^x_i = z_i|H_i, S^x_i, H_{-i}, Z^x_{-i})P(S^x_i|H_i, H_{-i}, Z^x_{-i}) \end{aligned}$$

and further simplified by noting

1. $P(Z^x_i = z_i|H_i, S^x_i, H_{-i}, Z^x_{-i}) = P(Z^x_i = z_i|S^x_i, Z^x_{-i})$ since emissions at a state of an HMM at a site are conditionally independent given the hidden state at the site

2. $P(S^x_i|H_i, H_{-i}, Z^x_{-i}) = P(S^x_i|H_i, H_{-i})$ since the hidden state, S^x_i, at site x does not depend on the allele in Z^x_{-i}

so that

$$P(Z_i^x = z_i|H, Z_{-i}^x) \propto \sum_{S_i^x} P(Z^x = z_i|S_i^x, Z_{-i})P(S_i^x|H_i, H_{-i}), \tag{II.8}$$

where the term $P(Z^x = z_i|S_i^x, Z_{-i}^x)$ denotes the emission probabilities of the model. The emissions can be given as

$$P(Z_i^x = z_i|S_i^x = k, Z_{-i}^x) = \begin{cases} 1-q & \text{if } z_i = z_k \\ q & \text{if } z_i \neq z_k, \end{cases} \tag{II.9}$$

where q may be thought of as a mutation probability, although it is expected to model sequencing errors more often than actual mutations.

The term $P(S_i^x|H_i, H_{-i}) = P(S_i^x = k|H)$ in eq. (II.8) is the posterior probability from the forward-backward algorithm under the Li and Stephens model using just the SNP haplotypes and, therefore, can be pre-calculated. For the purpose of prototyping convenience, in implementations a symmetrical "similarity matrix value" M_{ik}^x was used from Zhan Su's Gene Cluster method (Su, 2008) instead of $P(S_i^x = k|H)$. The M_{ik}^x entries of Zhan Su's symmetricised similarity matrices were defined as

$$M_{ik}^x = \begin{cases} \frac{\mathrm{P}(S_i^x=k|H)+\mathrm{P}(S_k^x=i|H)}{2} & \text{if } i \neq k \\ 0 & \text{otherwise.} \end{cases} \tag{II.10}$$

3.2 Inference of haplotype configurations

The first approaches considered for performing inference were centred around estimations of the posterior distribution of indel haplotype configurations $P(Z^x|H, R)$. Unfortunately, computing the likelihood $P(Z^x|H)$ is computationally inefficient. A correct factorisation of $P(Z^x = z|H)$ would be

$$P(Z^x = z|H) = P(Z_1^x|H)P(Z_2^x|H, Z_1^x)P(Z_2^x|H, Z_1^x, Z_2^x)\dots P(Z_{2N}^x|H, Z_1^x, \dots Z_{2N-1}^x). \tag{II.11}$$

The trouble with using this factorisation is that it depends on an ordering of the haplotypes (through the ordering of the Z_i^xs), which could influence the result. One solution would be to average over a small set of orderings, but that would have limited the reliability of the method, especially when considering large samples. Instead, the second triad approach was approximating the likelihood with a 'pseudo-likelihood' by

$$P(Z^x = z|H) \propto \prod_{i=1}^{2N} P(Z_i^x = z_i|H, Z_{-i}^x) \tag{II.12}$$

as mentioned in Section 3.1. Because this factorisation is not exact, an attempt was made to correct for the distortion this estimate gives.

The 'pseudo-likelihood' can be thought of as using 'too much information' in the sense that all but one of the conditional distributions included in the product condition on too many of the Z_i^x's. One theory is that this makes the factorisation too restrictive i.e. the probability distribution it represents is overly peaked around specific configurations. This bias was attempted to be resolved by decreasing the influence of the likelihood term by raising it to a power of λ (where $0 < \lambda < 1$). The result is the equation

$$\begin{aligned} P(Z^x|H,R) &\propto P(R|Z^x,H)P(Z^x|H) \qquad \text{(II.13)} \\ &\sim \prod_{l=1}^{N} P(R_n|g = \{Z^x_{2(n-1)+1}, Z^x_{2n}\}) \Big(\prod_{i=1}^{2N} P(Z_i^x = z_i|H, Z_{-i}^x) \Big)^{\lambda} \\ &\propto \prod_{l=1}^{N} P(R_n|g = \{Z^x_{2(n-1)+1}, Z^x_{2n}\}) \Big(\prod_{i=1}^{2N} \sum_{S_i^x} P(Z_i^x|S_i^x, Z_{-i}) P(S_i^x|H) \Big)^{\lambda} \end{aligned}$$

to calculate the posterior probability for any Z^x indel configuration in the search space.

The first corresponding inference method simply generated a large number of random candidate Z^x configurations according to GT posterior probabilities (normalised GT likelihoods) and calculated the posterior probability estimates (up to a sclaing factor according to eq. (II.12)) for all these candidate configurations. After evaluating all candidates, the configuration with the maximum posterior probability estimate was chosen to be the method's estimate of the global maximum. While there were a wide range of possibilities for improving this approach, development soon moved on to another approach for exploring the configuration space and directions based on calculating the 'pseudo-likelihood' were disregarded in the light of preliminary results (see the 'Naive' method in Section 3.2.2).

The next approach was to bypass the calculation of the likelihood by calculating conditional distributions of the Z_i^x configuration elements (the micro-configuration of haplotype pairs) based on other configuration elements. The conditional distributions of the model are easy to specify following this idea. For any individual n, the two relevant elements of Z^x will be the $k_1 = (\, 2(n-1)+1 \,)^{th}$ and $k_2 = (\, 2n \,)^{th}$ ones, for which the conditional distributions are

$$P(Z^x_{k_1}|H,R,Z^x_{-k_1}) \propto$$
$$P(R_l|g=\{Z^x_{k_1},Z^x_{k_2}\})\Big(\sum_{S_{xk_1}} P(Z^x_{k_1}|S_{xk_1},Z_{-k_1})P(S_{xk_1}|H_{k_1},H_{-k_1})\Big)$$
$$P(Z^x_{k_2}|H,R,Z^x_{-k_2}) \propto$$
$$P(R_l|g=\{Z^x_{k_1},Z^x_{k_2}\})\Big(\sum_{S_{xk_2}} P(Z^x_{k_2}|S_{xk_2},Z_{-k_2})P(S_{xk_2}|H_{k_2},H_{-k_2})\Big). \quad \text{(II.14)}$$

The accuracy of preliminary inference results (see Section 3.2.2) led to the choice of this latter approach.

3.2.1 Inference methods

There were three algorithms assessed in this work to explore the space of possible configurations for Z^x with the use of conditional distributions (eq. (II.14)).

A) Deterministic optimisation with Iterative Conditional Modes (ICM)

1. Pick an initial Z^x as a starting point.
2. Update iterations: for each individual in turn, set $Z^x_{k_1}$ and $Z^x_{k_2}$ to the values that maximise the two conditional distributions (eq. (II.14)). Each of $Z^x_{k_1}$ and $Z^x_{k_2}$ can be either 0 or 1, leading to 4 possibilities to choose from in each iteration.
3. Stop when no more changes are made. The result is the configuration corresponding to a local mode of the posterior.

The ICM algorithm (Besag, 1986) was the only deterministic approach explored in this work. The great advantage of ICM was that it found local maxima very fast, mostly in about 5 sets of iterations. While this speed would be desirable, the algorithm can not be expected to find the global maximum out of the numerous local maxima, and the fact that the search completely stops after finding a local maxmimum becomes a drawback. A remedy to this weakness could be to run ICM several times from different starting points, but then the task of deciding between the local modes would become a problem. To make such decisions requires the ability to compare configurations objectively, which leads back to the problem of using estimates of the whole posterior. Unfortunately, the incorrect pseudo-likelihood calculation did not prove applicable for this purpose, as the results demonstrated (Section 3.2.2).

B) Stochastic hill-climbing with pseudo-likelihood evaluation

1. Pick an initial Z^x as a starting point, calculate the posterior probability estimate of that point (based on eq. (II.13)) and set this as the best configuration thus far.
2. Update iterations: for each individual in turn, set $Z^x_{k_1}$ and $Z^x_{k_2}$ according to one of the micro configurations $00, 01, 10, 11$ randomly, according to the conditional distributions (eq. (II.14)). Calculate the posterior probability estimate at the end of each iteration round after updating the micro-configuration of last individual. If the result exceeds the maximum found thus far, save this configuration as the new maximum.
3. Stop after a set number of iterations. The result is the configuration found with the best posterior probability estimate throughout the entirety of the run. To calculate the posterior probability, the term $P(Z^x = z|H)$ is estimated with the pseudo likelihood (eq. (II.12)).

This algorithm is the stochastic version of ICM with the integrated use of the pseudo-likelihood-based approximate posterior probability evaluation.

C) Gibbs sampling

1. Pick an initial Z^x as a starting point.
2. Update iterations: for each individual in turn, set $Z^x_{k_1}$ and $Z^x_{k_2}$ according to one of the micro-configurations $00, 01, 10, 11$ randomly, based on the conditional distributions. Save each new configuration.
3. Stop after a set number of iterations (or when it is presumed that convergence is reached) and discard a set number of the first iterations (as burn-in iterations). The result is a sample of configurations that approximates the posterior distribution of configurations.

This algorithm is a proper MCMC-driven (Markov Chain Monte Carlo) simulation-based approach with many advantages over the previous alternatives. By producing posterior distribution samples, the use of pseudo-likelihood is not needed, while the whole search space of Z^x configurations can be explored in time. Moreover, with a sample at hand, it becomes possible to estimate the uncertainty of findings. Theoretically, the inference also shouldn't be stuck at local maxima.

The disadvantages of the Gibbs Sampler-based method are inherent to its MCMC nature in that it needs a fair number of iterations to explore the space of configurations (2^{2N} possibilities). It is not possible to know for sure when to stop the sampler (it is never fully certain whether it has converged).

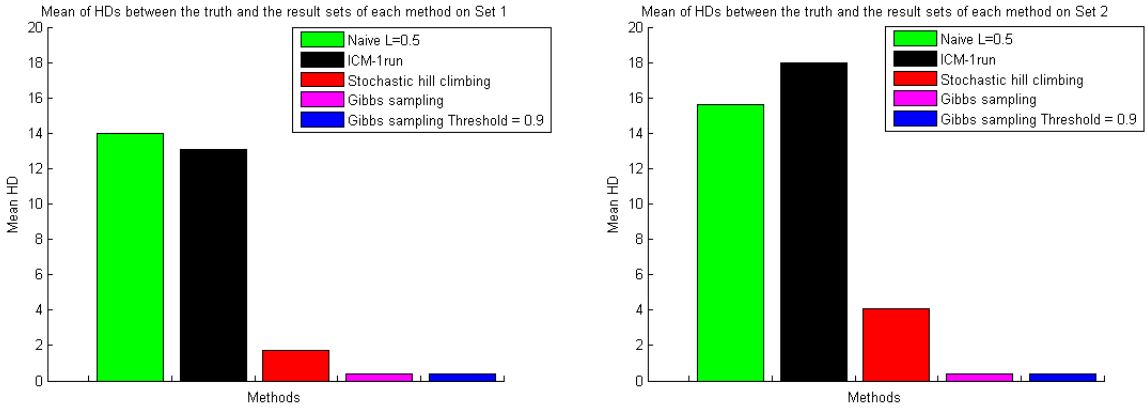

Figure II.7: Comparisons of performance between different inference methods on semi-synthetic data. For the purpose of this assessment, a set of SNPs were chosen to provide as a test set of 'pseudo indel' haplotypes. The bars show the accuracy of haplotype predictions using various inference approaches. The left side shows the accuracy of predictions with very informative synthetic GT probability data and the right side shows accuracy when the GT probabilities were more flat and less accurate. The error metric is the mean Hamming Distance (HD) between the true Z^x (pseudeo indel) vectors and the predictions of 5 runs on 10 pseudo-indel sites of 46 CEU HapMap individuals (92 haplotypes). 'Naive' refers to the independent sampling approach with $L = 0.5$ referring to the λ pseudo-likelihood suppressing parameter. The ICM method was performed with a single run and the Gibbs sampler was run with a threshold of 0.9 as well as without a threshold.

In an attempt to restrict the search space and speed up the algorithm, a threshold parameter was introduced. Whenever a GT posterior probability mode in the input was above this threshold, the Gibbs Sampler was allowed to make a deterministic choice favouring that GT during the updates.

3.2.2 Preliminary evaluation of inference methods, method choice

Due to the lack of any validation at the beginning of development, evaluations were restricted to synthetic and semi-synthetic data.

The setup of the semi-synthetic tests was the following: 10 SNPs were chosen randomly from the HapMap3 SNPs to serve as 'pseudo-indels' for each run. For the sake of fast comparison, the 'pseudo-indels' were assigned GTPs according to their original corresponding SNP GTs and also according to one of two different sets (a 'good' set and an 'ok' set) of presumed dataset quality characteristics. The sample size was 46, made up exclusively of CEU HapMap3 individuals.

Each GTP was seeded with an arbitrary value (with the highest probability assigned to the true SNP GT value) which were then perturbed randomly, to simulate uncertainty, again according to presumed dataset characteristics. For instance, the 'pseudo-indel' GT probabilities corresponding to the true SNP GTs in the good set were set, rather optimistically, to

0.83 and 0.73 in the 'ok' set. The probability values were then perturbed by adding values drawn from $U(-0.17, 0.17)$ and $U(-0.21, 0.21)$, respectively, for the two sets. Looking back, it is clear that simulating of the data this way was very crude and did not reflect the actual indel GT data (the maximum probability in the indel GT data would be 0.6 at most). Figure II.7 compares accuracy results across methods, clearly suggesting the superiority of the Gibbs Sampling based method. In spite of the limitations of this analysis, the results seemed convincing enough to support the decision of singling out the Gibbs Sampler as the method of choice.

3.2.3 Inference initialisation

A part of the effort in method development was put into exploring possibilities for fast and simple initialisation, which seemed to have a considerable effect on convergence speed. The below initialisation methods pick GTs 0, 1 or 2 for each individual in different ways, but the initial phase of heterozygous GTs is chosen randomly in each case with equal probabilities ($P(HT = 1, 0|GT = 1) = P(HT = 0, 1|GT = 1) = 0.5$).

The simplest way to generate starting GTs is assigning them at random with equal probabilities on the GTs ($P(GT = 0) = P(GT = 1) = P(GT = 2) = 1/3$) for each genotype for each individual. This method of initialisation was used mainly for robustness and convergence testing.

A reasonable choice for initialising the algorithms was to pick the starting Z^x's according to the input GTL modes. Another simple solution was to set all the GTs to be 0, since a relatively high proportion of the indel GTs can be expected to be homozygous reference.

A more sophisticated initialisation method produced GTs according to both the input GT likelihoods and corresponding variant allele frequency estimates found using an Expectation-maximisation (EM) setup. The EM algorithm used for finding the ML estimate of the indel variant allele frequency θ_x for indel x maximises

$$L(\theta_x) = \prod_{l=1}^{N} P(R|\theta_x) = \prod_{l=1}^{N} \sum_{g=0}^{2} P(R_{xl}|G_{xl} = g) 2^{I_{\{g=1\}}} \theta_x^g (1 - \theta_x)^{2-g} \tag{II.15}$$

in the following manner:

1. Initialisation: set $\theta_x^{(t)}$ randomly $\sim$U(0,1)

2. E-step: for the current $\theta_x^{(t)}$ calculate

$$\begin{aligned} P(G_{xl}|R_{xl},\theta_x^{(t)}) &\propto P(R_{xl}|G_{xl})P(G_{xl}|\theta_x^{(t)}) \\ &= P(R_{xl}|G_{xl})2^{I_{\{g=1\}}}\theta_x^g(1-\theta_x)^{2-g} \end{aligned}$$

   where $P(R_{xl}|G_{xl})$ comes from the indel genotype likelihoods by normalisation.

3. M-step:

$$\theta_x^{(t+1)} = \frac{\sum_{l=1}^{N}\sum_{g=0}^{2} g * P(G_{xl}|R_{xl},\theta_x^{(t)})}{2N}$$

As normal, the EM algorithm had to be run multiple times to better allow the multiple runs to collectively find a global maximum of the likelihood or some high maxima. After finding a suitable allele frequency, the generation of genotypes for each individual was done by drawing genotype values randomly according to $P(G_{xl}|R_{xl},\theta_x^{(t)})$.

Table II.2
Comparison of initialisation methods. The Sequenom validation data was taken to be the truth for calculating these error rates. These values are typical and are based on the observation of over 20 runs each.

Method	Typical error rate of init method
Uniform on {0,1,2}	66.67%
All hom. ref. (0)	30.49%
EM for allele freq. optimisation	∼24%
Dindel GTL modes on {0,1,2}	19.69%

Table II.2 shows typical rates of disagreement (on all sites and individuals) between GTs generated by each method and the calls of the Sequenom validation data of 90 sites and 59 individuals (see the dataset setup used for validation in Section 4.1.2). The completely random GT initialisation has an error rate of 2/3 as expected, and the $\forall\ GT = 0$ method does relatively well due to the ∼70% hom. ref. frequency in the validation data. Following this comparison, the initialization of most phasing runs was by either the EM algorithm or simply at the input modes.

4 Results and discussion

The inference methods presented in Section 3.2 for the SLVC model (Section 3.1) were implemented in MATLAB. For simplicity, the implementation used the *lstree* module from Zhan Su's GENE CLUSTER method (Su, 2008) for pre-calculating local similarity between haplotypes, based on the phased input.

The pipeline used for SLVC runs consisted of the following steps:

- Start with a dataset of indel GTLs and positions, a dataset of phased low-uncertainty SNP haplotypes and positions and a corresponding recombination map.
- Pre-calculate similarity matrices for the position of each indel based on the SNP haplotype data and the recombination map.
- Perform SLVC runs with the use of indel GTPs (calculated from the GTLs) and the pre-calculated similarity matrices to produce samples corresponding to the posterior probability of indel configurations.
- Calculate indel HT and GT posterior probabilities and calls based on the MCMC samples for each indel.
- Evaluate results.

4.1 Results using indel data from the 1000 Genomes Project

The first set of evaluation experiments were performed using real indel data from the 1000 Genomes Project in the absence of phased indel validation. Accuracy assessments proceeded with comparing SLVC GT call results to the unphased Sequenom GT call set (see Section 2). The characteristics of these data are explored in Section 2.1.

4.1.1 Preliminary experiments

The indel validation set where Dindel and Sequenom agreed in both position and variation consisted of 96 indels on chromosome 20 for 59 CEU HapMap individuals. Similarity matrices were pre-calculated for each indel locus using 30,038 HapMap3 SNPs on chromosome 20 of the same 59 individuals.

Following some initial assessment, the validation set was filtered to exclude 6 indels. Four sites were excluded due to their high missing data rate in the Sequenom set: site 6 at position 3190311, site 40 at position 24673109, site 42 at position 28238554, and site 89 at position 59351366. Two more sites were excluded because corresponding data seemed

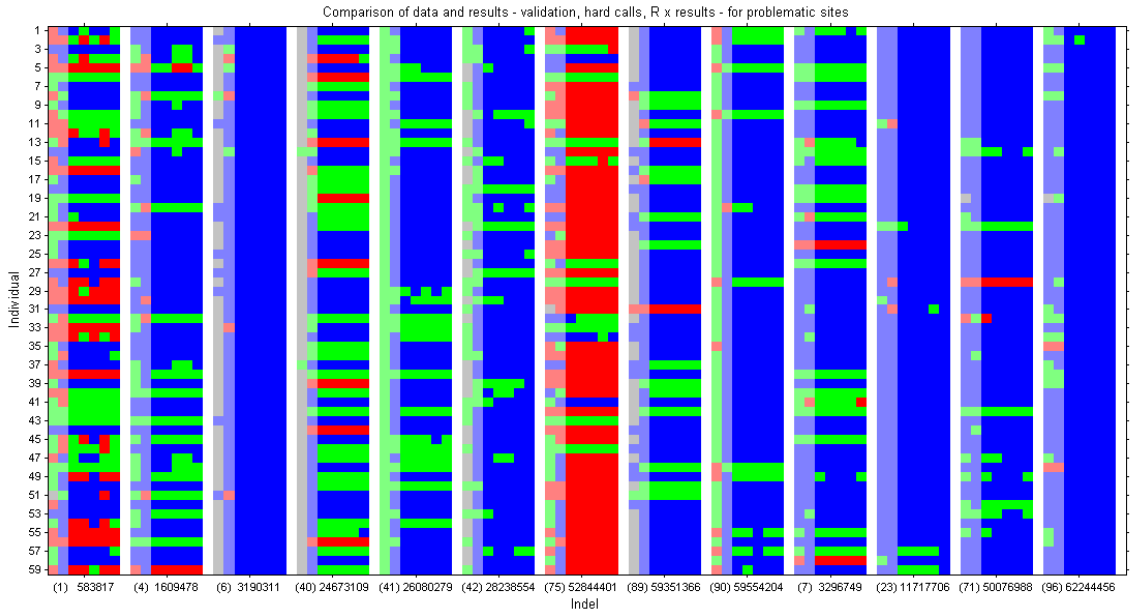

Figure II.8: Problematic and 'hard' sites. This plot presents the difference between corresponding Sequenom GT calls, Dindel GT modes and five sets of SLVC GT predictions for each site and each individual. The presented sites were chosen because of high missing data rate and high disagreement between the Dindel and Sequenom sets. Each major column corresponds to an indel. The first two sub-columns correspond to the Sequenom and Dindel calls, while latter five sub-columns correspond to GTs found by five different SLVC runs. The colours of boxes in the sub-columns refer to GTs (blue - homozygous reference, green - heterozygous, red - homozygous variant and grey refers to missing data).

erroneous. Site 41 at position 26080279 was excluded because all GTs in the Sequenom data were called heterozygous for this indel, which is highly unlikely to be correct (according to Hardy Weinberg equilibrium assumptions to say the least). Site 90 at position 59554204 was excluded for two reasons: first, there seemed to be a high rate of disagreement between the Dindel and the Sequenom data, and second, because preliminary phasing results suggested a possible 0 to 1 and 1 to 2 mislabelling problem at this site (which may be due to mislabelling just on one of the strands). Figure II.8 presents Sequenom GT calls and SLVC predictions for the 13 most problematic-looking sites.

After filtering the validation data, a few experiments were performed for tuning the SLVC method for setting auto-call thresholds, initialisation schemes, burn-in and run length of the Gibbs Sampler (see Section 3.2 for inference methods). The experiments showed that using different burn-in iterations did not affect results significantly, while using different initialisations affected the number of iterations required to achieve accurate predictions (as mentioned in Section 3.2.3). Thresholding also did not matter in practice, since all GTPs (normalised from the GTLs) in the Dindel input set were below 0.6 (see Figure II.2) and setting thresholds below 0.6 led to very low accuracy, as expected (data not shown).

4.1.2 Indel GT call validation

The third set of experiments were performed with 5 independent SLVC runs of 10,000 iterations. Runs were initialised according to the input GTPs (see Section 3.2.3). SLVC accuracy was characterised by comparing both GT probabilities and GT calls to hard Dindel calls and Sequenom calls. Robustness and convergence were assessed based on the agreement of results between different runs.

GT posterior probabilities were calculated from haplotype probabilities as

$$\begin{aligned} P(GT_k^x = 0|H,R) &= P(Z_{k1}^x = 0, Z_{k2}^x = 0|H,R) \\ P(GT_k^x = 1|H,R) &= P(Z_{k1}^x = 0, Z_{k2}^x = 1|H,R) + P(Z_{k1}^x = 0, Z_{k2}^x = 1|H,R) \\ P(GT_k^x = 2|H,R) &= P(Z_{k1}^x = 1, Z_{k2}^x = 1|H,R), \end{aligned}$$

and hard SLVC GT calls were made simply by taking the most probable GTs for each individual at each site. Figure II.9 presents for each site: Sequenom GT distributions; the disagreements between Dindel modes and Sequenom calls; and GT miscall rates for SLVC predicitons (compared to the Sequenom data). Comparing Figures II.9 E and II.9 F to Figures II.9 C and II.9 D, it is apparent that GTs predicted by SLVC agree with the Sequenom calls significantly more than Dindel modes. This trend is observable for all three classes of GTs (hom.ref., het. and hom. var.), and the overall miscall rate dropped to 8.2% using SLVC compared to the 19.5% disagreement of predictions with Dindel GTL modes. It is also clear that miscalls of heterozygous GTs contribute to the majority of errors for both Dindel and SLVC. Since the distribution of Sequenom GT classes is uneven, miscall rates were stratified by Sequenom GT values as shown in Figures II.9 G and II.9 H. This plot shows that GTs 1 and 2 are miscalled much more frequently than homozygous reference GTs. The Figures further show that there is a striking correlation between the overall Dindel miscall rates and the variance of Sequenom GTs, while the correlation between SLVC miscall rate and GT variance is lower.

The comparison of confusion matrices corresponding to SLVC results and Dindel modes (see Table II.3) prove that SLVC GT predictions provide significant improvements in all cases except for a slightly increased $GT_{validation}=2$ to $GT_{phasing}=1$ miscall rate.

The overall improvement is further demonstrated by Figure II.10. The scatter plot of miscall rates per site (A) relates SLVC results to those with Dindel GTL modes. The heat plot (B) presents overall miscall rates from five independent SLVC runs and discordance rates between the runs. From these miscall and discordance rate figures it appears that there is a

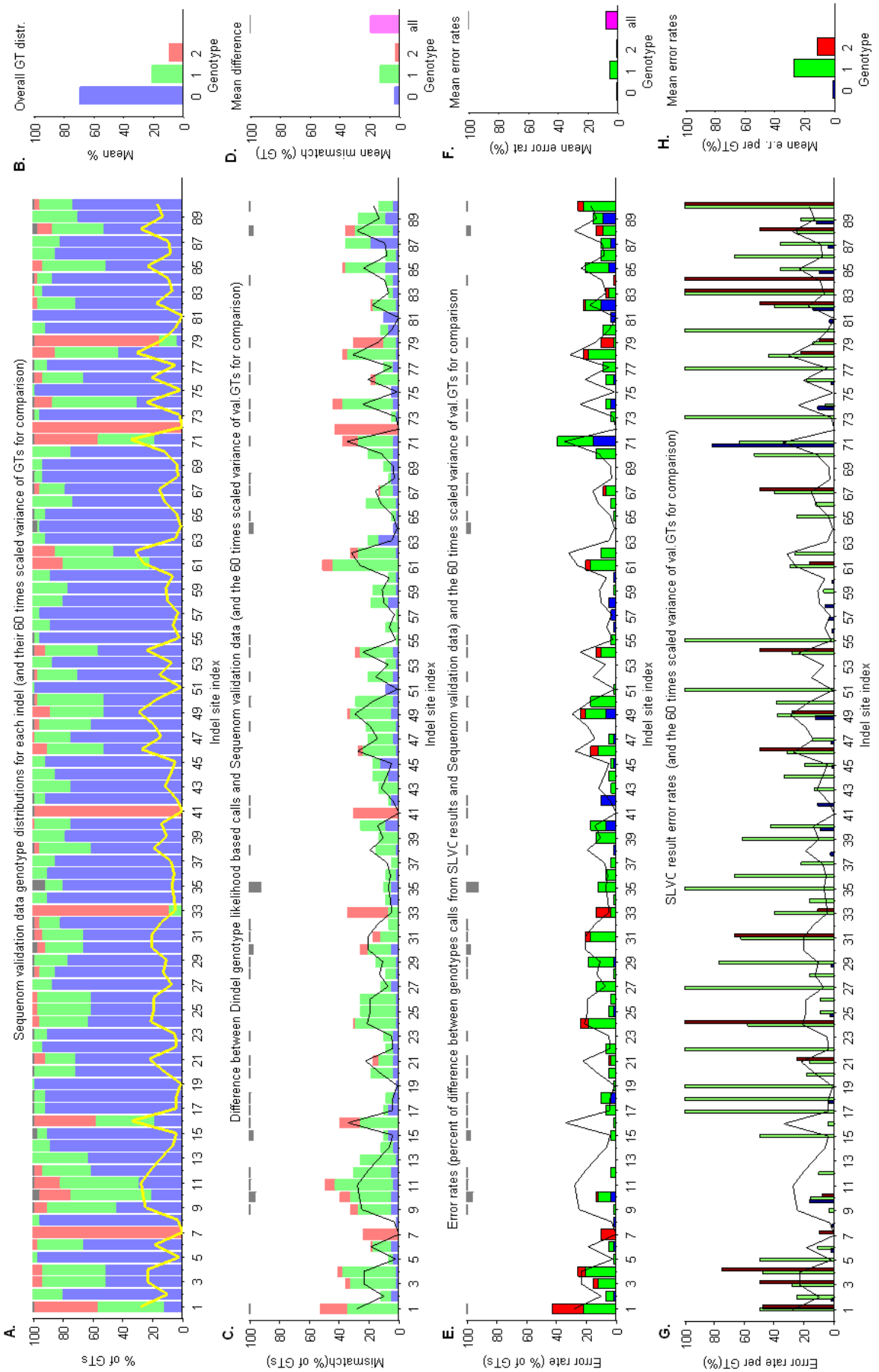

Figure II.9: Sequenom calls, Dindel GTL modes and SLVC predictions. The top row presents (A) the distribution of Sequenom GTs for each site, and (B) the overall distribution of GTs. The second row presents (C) the degree of discordance between Dindel GTL modes and Sequenom calls for each site per GT and (D) the mean discordance per genotype. Similarly, the third row presents (E) discordance between SLVC predictions and Sequenom calls per site per GT and (F) the genotype discordance means. The fourth row presents (G) the percentage of the hom.ref., het. and hom.var. GTs predicted by SLVC to be different from the Sequenom calls. Here 100% corresponds the number of 0's, 1's or 2's in the Sequenom set. (H) shows mean miscall percentages for each genotype. The yellow and black solid lines show the variance of GTs in the validation data; the colouring of bar graphs refers to GTs as follows: blue - hom. ref., green - het. and red - hom. var. Grey bars show the percentage of missing data in the Sequenom set.

Table II.3: Confusion matrices. Comparison of genotype call differences between SLVC results and the Sequenom data (bottom); confusion matrix for Dindel modes on the same set of sites (top). The 4^{th} column and 4^{th} row in each table present row and column sums. The 5th row presents the number of missing genotypes in the validation data and row 6 shows the total number each genotype is called.

		Dindel GTs (modes)			
		Hom.ref.(0)	Het.(1)	Hom.var.(2)	Sum
Sequenom GTs	Hom. reference (0)	3474 (66%)	**98 (1.9%)**	**85 (1.6%)**	3657 (69.5%)
	Heterozygous (1)	**513 (9.8%)**	416 (7.9%)	**178 (3.4%)**	1107 (21.0%)
	Hom. variant (2)	**139 (2.6%)**	**23 (0.4%)**	335 (6.4%)	497 (9.4%)
	Sums	*4126 (78.4%)*	*537 (10.2%)*	*598 (11.4%)*	*5261 (100%)*
	Untyped/missing	*33*	*9*	*7*	*49*
	Sums incl. untyped	*4159*	*546*	*605*	*5310*

		SLVC-predicted GTs			
		Hom.ref.(0)	Het.(1)	Hom.var.(2)	Sum
Sequenom GTs	Hom. reference (0)	3594 (68.3%)	**52 (1%)**	**11 (0.2%)**	3657 (69.5%)
	Heterozygous (1)	**259 (4.9%)**	799 (15.2%)	**49 (0.9%)**	1107 (21.0%)
	Hom. variant (2)	**18 (0.3%)**	**43 (0.8%)**	436 (8.3%)	497 (9.4%)
	Sums	*3871 (73.6%)*	*894 (17%)*	*496 (9.4%)*	*5261 (100%)*
	Untyped/missing	*34*	*11*	*4*	*49*
	Sums incl. untyped	*3905*	*905*	*500*	*5310*

significantly lower disagreement (3.4% disagreement rate) between the GT results of different SLVC runs then between the SLVC GT results and the Sequenom data ($\sim$ 8.2% error rate). In fact, the SLVC results are closely clustered together and slightly further apart from the validation data GTs. Hence, SLVC seems to be relatively consistent across multiple runs, but it seems to consistently converge to indel configurations that are slightly different from the configurations suggested by the Sequenom calls.

If the stationary distributions of the Gibbs Sampler in SLVC correspond to results that are slightly different from the validation data GTs, that may well be the result of the significant disagreements between the input and validation data - in other words the reliability of the indel GTLs.

As a next step, the effect of the reliability of GTL inputs on SLVC accuracy was examined. The numbers of correct and incorrect GT calls were counted for both the SLVC predictions

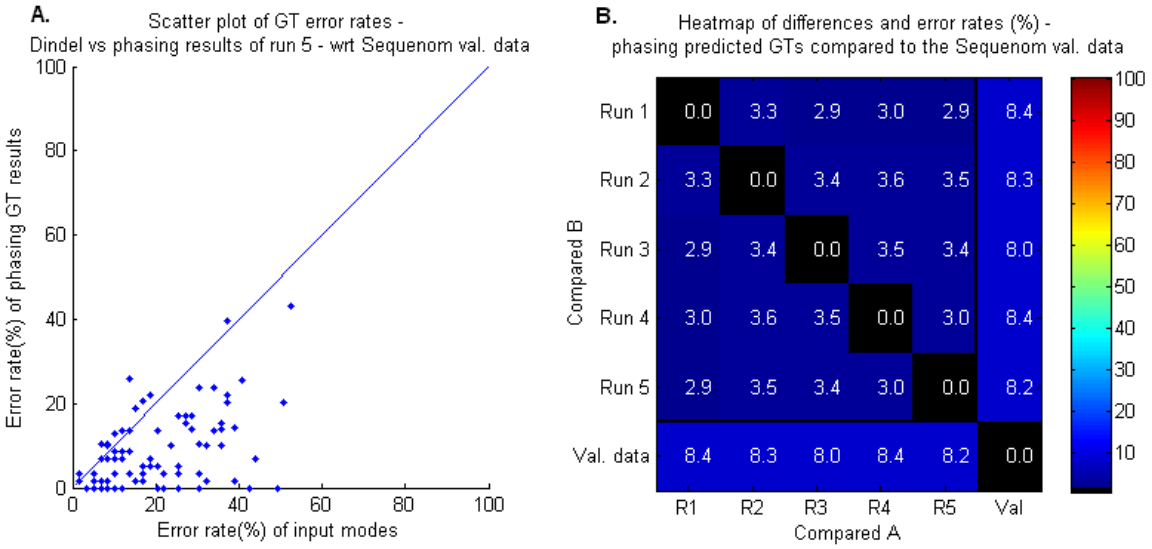

Figure II.10: SLVC prediction improvements and consistency. (A) Scatter plot of Dindel GT modes and GTs from phasing results from run 5. (B) Heat map of GT disagreements between 5 phasing run results and the Sequenom validation data. All 5 result sets were acquired by the Gibbs Sampling method (see Section 3.2.1) with a threshold of $t = 0.9$, initialised by the EM method (see Section 3.2.3) and run for 10000 iterations, from which the first 2000 were excluded as burn-in.

Table II.4: The relationship between input GT mode errors and phasing predicted GT errors. All GTs were compared to the Sequenom validation data.

	Input mode GT call incorrect	Input mode GT call correct
SLVC miscalls	299	133
Correct SLVC calls	737	4141

and the Dindel modes, as shown in Table II.4. With these figures at hand, the following empirical probability and conditional probability estimates were compared:

$$\begin{aligned} \mathrm{P}(GTerror) &= \frac{299+133}{299+133+737+4141} = 0.081 \\ \mathrm{P}(GTerror \mid misleading\ input) &= \frac{299}{299+737} = 0.288 \\ \mathrm{P}(GTerror \mid helpful\ input) &= \frac{133}{133+4141} = 0.031 \end{aligned} \tag{II.16}$$

The error rate estimates clearly show that although the SLVC algorithm's predictions present an improvement compared to the original Dindel GT probability modes, the method's accu-

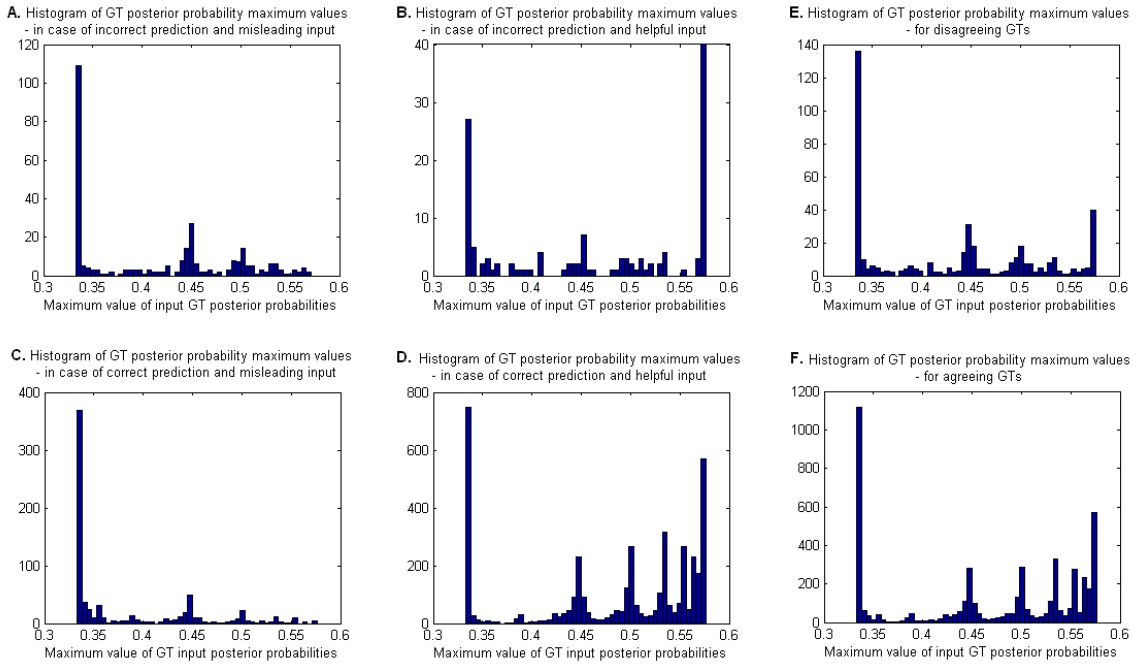

Figure II.11: Histograms representing the distributions of maximum posterior probabilities (normalised from the input GT likelihoods), stratified by prediction errors. (A), (B), (C) and (D) correspond to prediction result pairs of phasing error with misleading input, phasing error with helpful input, correct phasing with misleading input and correct phasing with helpful input, respectively, similarly to the arrangement of Table II.4. (E) and (F) represent the marginal distributions for the cases of phasing SLVC miscalls and correct calls.

racy heavily relies on the accuracy of the input. Based on these results, it is reasonable to assume that the reason why the SLVC results consistently converge (with ∼3.1% difference between different runs) to configurations slightly more away from the truth may be because to the imperfections of the input data. Results can be expected to improve with better GTL input, but SLVC can also greatly improve the reliability of GTL sets.

A more detailed analysis of how the reliability of inputs effects SLVC results was performed including information on the 'flatness' of likelihoods. Figure II.11 presents the distribution of the highest Dindel GT probabilities stratified by correctness and SLVC prediction accuracy. The histograms show that SLVC GT prediction errors occur mostly when the GTLs are flat (sub-figure A, B, E), but SLVC is also able to pool information from other individuals in order to determine the GT correctly, even if the GTLs are flat (sub-figure C). The relatively helpful characteriszation of uncertainty in the Dindel GTLs allowed SLVC to assess the accuracy of the input most of the time (sub-figure D) but SLVC sometimes also seems to have disregarded correct GTLs even if the probabilities were high (sub-figure B). Overall, SLVC miscalls tend to occur when the input GTLs are completely flat or when they are misleading.

Verdict

When the purpose of running the method is to improve the reliability of reference datasets, then the results are adequate and SLVC is clearly valuable in that respect. However, if the purpose of running SLVC is to phase a GT dataset, it is advisable to provide the method with reliable GTLs.

4.1.3 Explorations of temporal behaviour

Following the doubts raised (by findings shown in Figure II.10) regarding the convergence properties of SLVC, evaluation of the method proceeded with exploring the temporal behaviour of the method. SLVC was run five times and multiple error measures and scores were calculated from sub-samples at a number of tracking points for each run. The measures included simple miscall rate between hard GT and HT call results, the root mean square deviation between GT and HT probabilities and various 'proper' scores.

The proper scoring rules used for assessing convergence were:

- Logarithmic score (L): $L = -log(\mathrm{p}_{truth}) \in (-\infty, 0]$
- Brier score (B): $B = 1 - 2\mathrm{p}_{truth} + \mathbf{p} \cdot \mathbf{p} \in [0, 2]$
- 1-Spherical score (1-S): $1 - S = 1 - \frac{\mathrm{p}_{truth}}{\sqrt{\mathbf{p} \cdot \mathbf{p}}} \in [0, 1]$

where $\mathbf{p}$ is the vector of posterior probabilities ($\mathrm{p}_0 = \mathrm{P}(GT{=}0)$, $\mathrm{p}_1 = \mathrm{P}(GT{=}1)$ and $\mathrm{p}_2 = \mathrm{P}(GT{=}2)$) predicted by SLVC and p_{truth} is the probability assigned to the true value (the truth according to the Sequenom data). Naturally, the different scoring rules have different properties that make them useful for different purposes. The logarithmic score is widely used because it is essentially the likelihood and it has the so-called 'local' property of depending only on the probability assigned to the correct GT. One of its disadvantages is that it is hard to interpret the meaning of its unbounded score values outside of comparison contexts. The Brier and Spherical scores on the other hand are not local, but have the 'effective' property which means that the application of the scoring rules 'encourage' p to be close to the true probabilities, which may be beneficial when assessing the accuracy of SLVC results (Bickel, 2007). The scores were calculated for all SLVC GT prediction vs. Sequenom call pairs and were aggregated (the mean calculated across sites and individuals) for overall tracing.

Various trace plots are presented in Figure II.12, with focus on convergence, consistency and accuracy over time and the effect of burn-in. The plots show that the results of different runs converge but convergence slows down after around 3000 iterations (see plots E-I). Despite

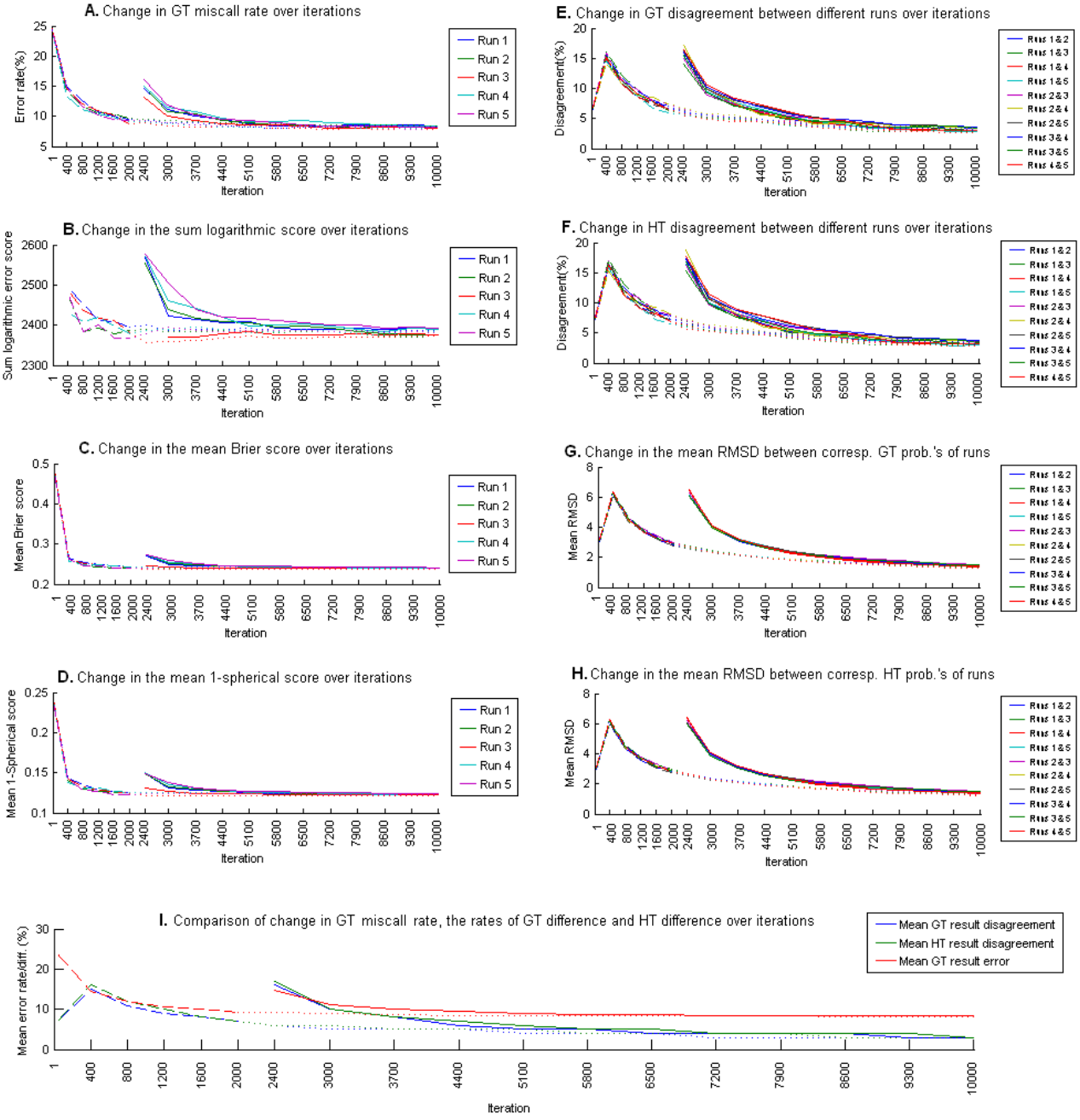

Figure II.12: Trace plots using results from 5 SLVC runs over 10,000 iterations. Plots on the left (A-D) compare intermediate SLVC results to validation data. Plots on the right (E-H) compare intermediate results between SLVC runs. Plot (I) presents traces with means across runs from plots A, E and F. In all subplots, dashed lines correspond to the first 2000 iterations, dotted lines correspond to results based on complete samples after 2000 iterations. Solid lines correspond to results based on samples discarding 2000 iterations as burn-in. (A) GT error rates of 5 runs (% of all difference compared to validation data). (B) Sum of Logarithmic scores over individuals and sites. (C) Mean Brier scores over GT probabilities of all individuals for all sites. (D) Mean 1-Spherical scores over GT probabilities of all individuals for all sites. (E) GT disagreement between 5 runs (% of all difference between each pair). (F) HT call disagreements between 5 runs (% of all difference between each pair). (G) Mean root mean square difference (RMSD) between found GT probabilities over GTs of all individuals for all sites. (G) Mean RMSD between found GT probabilities over HTs of all individuals for all sites. (I) Comparison of A, E and F: Mean GT error rate, mean GT disagreement and mean HT disagreement.

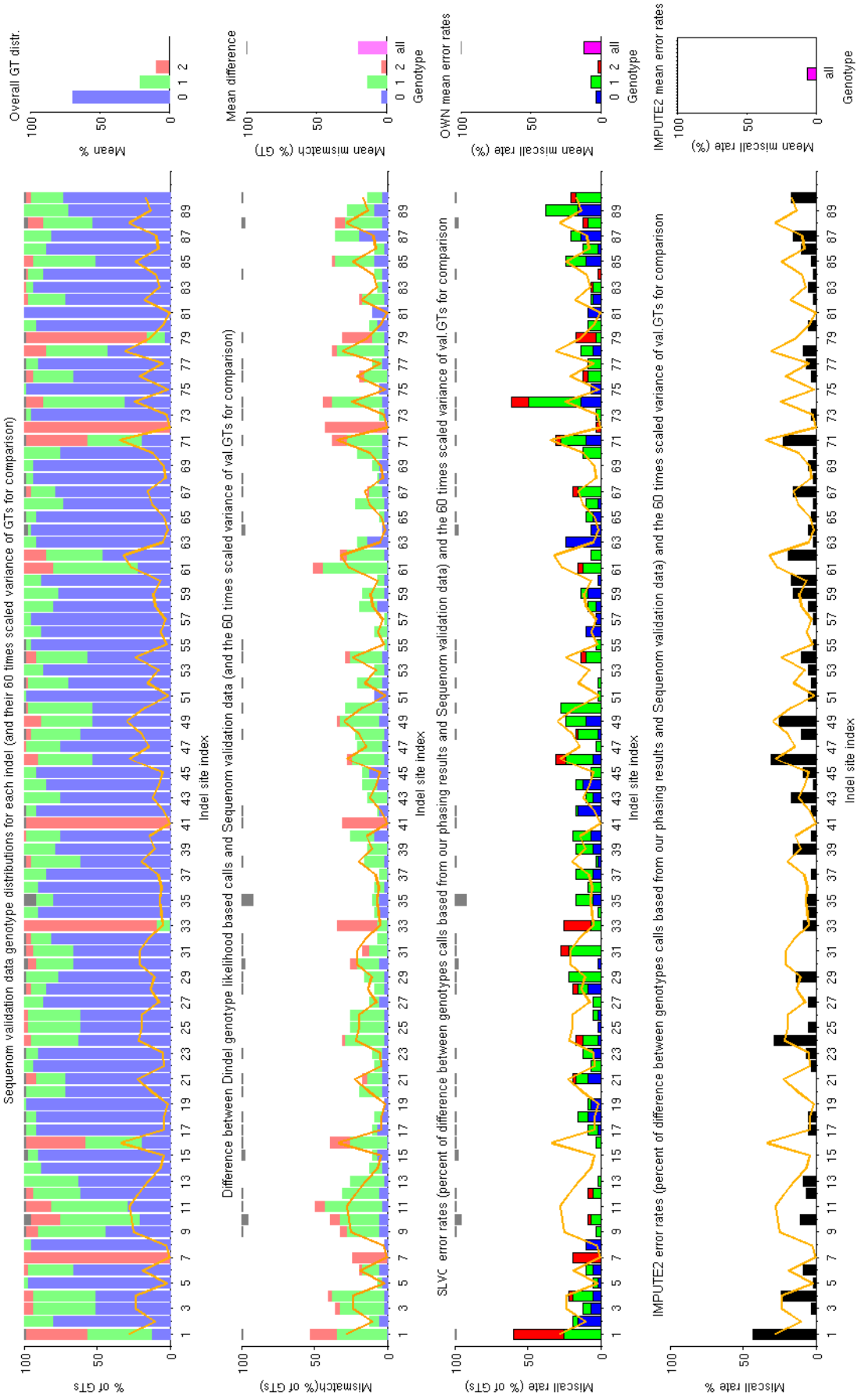

Figure II.13: Comparison of SLVC GT prediction results with IMPUTE2 GT results using indel data. The first two rows describe the dataset and the third row presents results of SLVC as before (in Figure II.2). The last row presents the IMPUTE2 results (these results are not scaled according to the presence of NaN values as the results in other rows).

the continuation of slow convergence until the end of the experiments, improvements in the GT error rate and proper scores stall completely around 2000 iterations. It can also be observed that discarding the first 2000 iterations does not improve the final results. Comparison of these overall trends (Figure II.12I) agrees with previous observations that SLVC converges towards a stationary distribution that would correspond to the inference of GT indel configurations that do not fully agree with the validation data GTs. Proper convergence analysis is presented in Section 4.2.

4.1.4 Comparisons with IMPUTE2 using real indel data

A set of comparisons were performed between SLVC and the IMPUTE2 method (Howie et al., 2011, 2009). The first comparison was performed using the same indel datasets as in the previous sections (Dindel GT likelihood data, Sequenom validation data and a HapMap3 SNP scaffold). IMPUTE2 was run with default settings (an effective population size N_e=20,000, 500kb buffer regions and 100 iterations of which 10 were used as burn-in) for making calls based on probabilistic input (GTPs) and the use of a haplotype scaffold as a reference panel.

The results (Figure II.13) show that IMPUTE2 achieves slightly better accuracy with only a 6.04% miscall rate, while SLVC achieved a 9-12% miscall rate. The accuracy results were also directly compared (Figure II.14) leading to the conclusion that SLVC does not fall far behind IMPUTE2 in terms of accuracy despite the fact that it performed inference on variants independently.

4.2 Convergence analysis

The consistency of results between different SLVC runs (see Section 4.1.2 and in particular, Figure II.10) and the experimental results on temporal behaviour (see Section 4.1.3 and Figure II.12) suggest that overall SLVC consistently converges in 2-6,000 iterations and may be expected to provide reasonable results even in about 400 iterations. However, previous analyses only explored overall convergence and not each data point (GTs per individual per loci) individually. Convergence experiments were set up particularly to assess convergence properties quantitatively. These experiments took into account the minor allele frequency (MAF) of different variants.

The convergence experiments used 86 indels of 59 individuals, which amounted to 5,074 GTs in total. The indels were divided into two groups, rare (MAF$<$5%) and common (MAF$\geq$5%). There were 62 common and 24 rare indels found, resulting in 3,658 common GTs and 1,416 rare GTs in total.

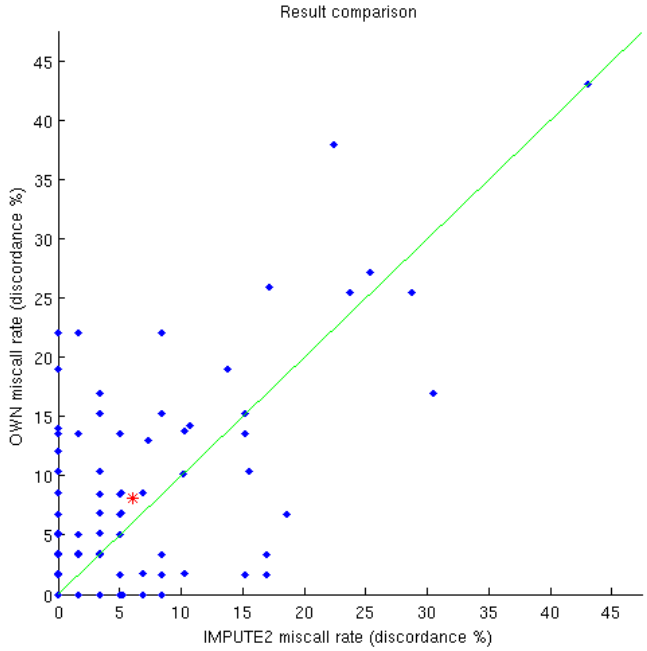

Figure II.14: Scatter plot of GT miscall rates achieved with SLVC and IMPUTE2. The red star corresponds to the mean miscall rates of the two sets.

Twenty independent chains were run with over-dispersed (completely random) starting points to measure convergence. Results were obtained for a set of different burn-in settings ($B \in \{$ 0, 100, 200, 300 $\}$) and were recorded at different iterations ($I \in \{$ 50, 100, 200, 400, 600, 800,1000, 1200, 1400 $\}$, meaning the sample size, the number of iterations after burn-in), resulting in 36 assessment points for analysis.

The $\hat{R}$ metric of Gelman and Rubin (Gelman et al., 2003) was calculated for all GTs for all 36 assessments. This convergence metric is a ratio of some output's variance inside and variance between a number of MCMC chains. Values close to 1 indicate good convergence. For the purpose of using $\hat{R}$, GTP prediction triplets were each converted into expected GT counts or 'dosages' (as in Howie et al. (2009)) as

$$d = \sum_{g=0}^{2} g\, p_g \tag{II.17}$$

with $d \in [0, 2]$, $g \in \{0, 1, 2\}$ referring to GT and p_g referring to the probability inferred for GT g. Originally, $\hat{R}$ was designed for normally distributed estimands. While the normality of dosages can not be guaranteed, the $\hat{R}$ is expected to be informative for assessing the convergence of SLVC, too.

Figure II.15 presents $\hat{R}$ histograms of all 5,074 $\hat{R}$ measurements for all assessment points and Figure II.16 presents percentages of $\hat{R}$ measurements that exceed convergence thresholds. It

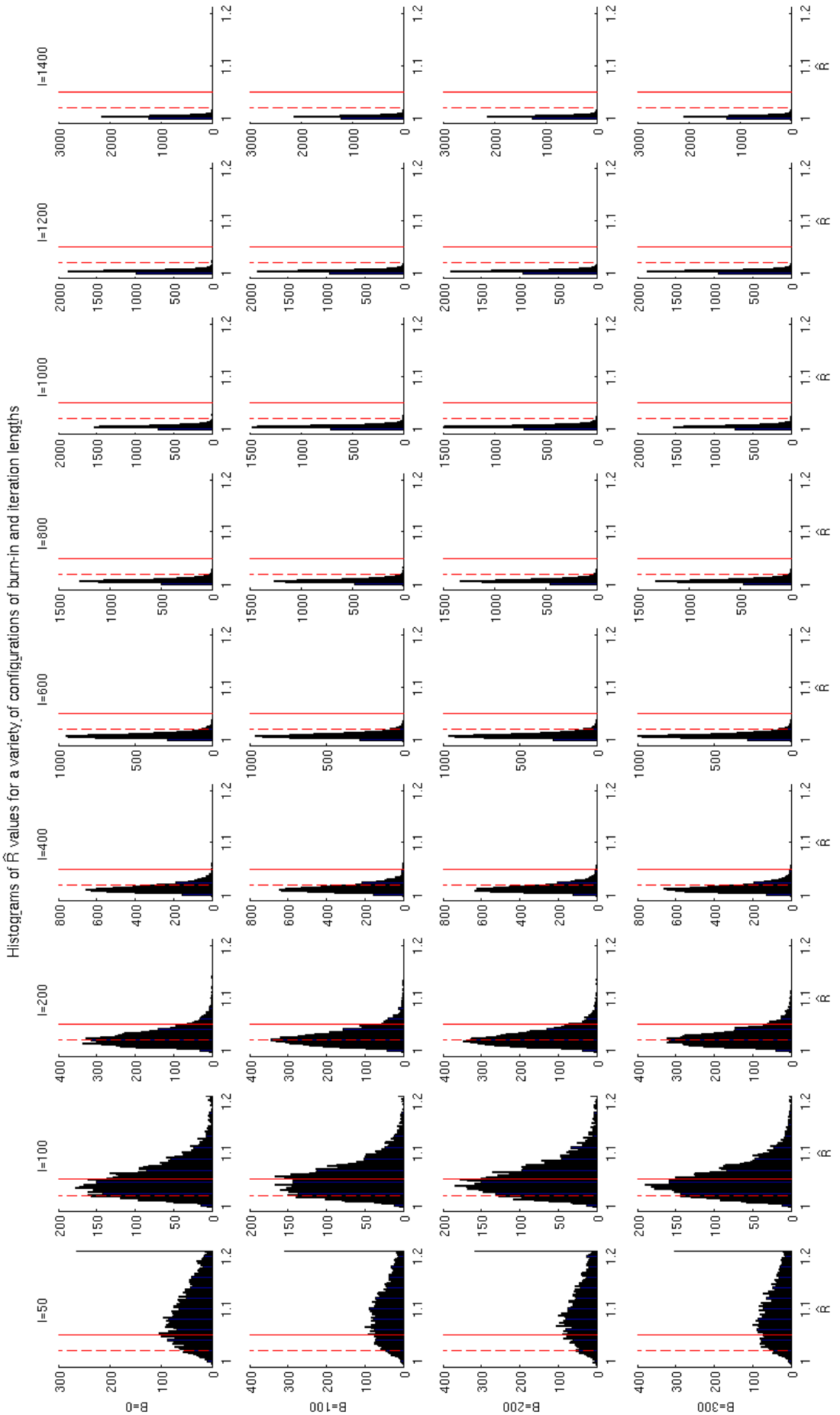

Figure II.15: Histograms of $\hat{R}$ measures for all 5,074 genotypes for all assessment points. Each row corresponds to one burn-in setting and each column corresponds to one stage (iteration) of the runs. The solid vertical red line corresponds to an $\hat{R}$ measure of 1.05, the dashed red line corresponds to an $\hat{R}$ measure of 1.02. The results were obtained from running 20 independent chains with completely random initialisations.

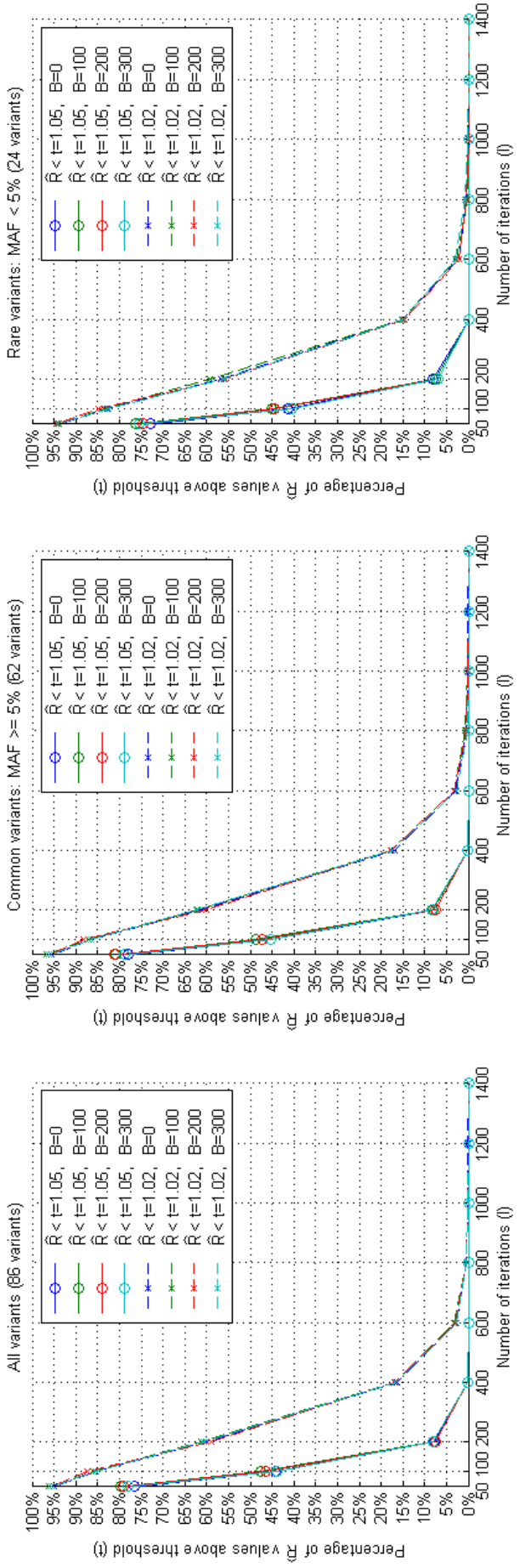

Figure II.16: Percentages exceeding thresholds: percentages of GTs for which the $\hat{R}$ measures exceed 1.02 (dashed lines) and 1.05 (solid lines) for each burn-in setting. The three subplots show results for all (left), common (middle) and rare (right) indels.

is important to note that the convergence threshold $\hat{R}$<1.05 is considered reasonably strict and the $\hat{R}$<1.02 is considered very strict.

Table II.5 concludes the convergence analysis by presenting mean percentages of $\hat{R}$ measurements that exceed the convergence threshold 1.02 for different MAFs, burn-in and sample sizes. Although the results may be subject to errors when estimating the variance, the percentages provide strong evidence that increasing burn-in has negligible effect on the $\hat{R}$ measures, while collecting larger samples decreases $\hat{R}$ drastically for both rare and common variants.

Table II.5: A breakdown of convergence results by burn-in, sample size and MAF. The figures present the percentage of $\hat{R}$ measurements that exceed the 1.02 threshold.

Common SNPs		Sample size				
		200	400	600	800	1000
Burn-in iterations	0	0.61	0.17	0.03	0.00	0.00
	100	0.62	0.18	0.03	0.01	0.00
	200	0.60	0.18	0.03	0.01	0.00
	300	0.62	0.17	0.03	0.00	0.00

Rare SNPs		Sample size				
		200	400	600	800	1000
Burn-in iterations	0	0.57	0.15	0.02	0.00	0.00
	100	0.59	0.15	0.03	0.00	0.00
	200	0.56	0.15	0.02	0.00	0.00
	300	0.56	0.15	0.03	0.01	0.00

Verdict

The results indicate that for the used indel dataset, SLVC converged reasonably well (with almost all $\hat{R}$ measurements below 1.05) in about 300-400 iterations and converged very well ($\hat{R}$<1.02) in 600-800 iterations from completely random initialisations, irrespective of any burn-in and irrespective of the MAF of indels. Naturally, with the use of better initialisations, SLVC is expected to reach convergence faster in real applications.

Limitations

It is important to note that this convergence test was performed with a rather small number of individuals compared to what is expected in realistic datasets. However, there is little reason to assume that convergence would be much slower for larger data sets - clearly, there would be a larger search space to cover, but there would also be more information for the

algorithm to use in order to improve results in every iteration. The convergence tests also showed, similarly to previous analyses, that using burn-in does not have much affect, but achieving adequate sample size is important for the convergence of the algorithm.

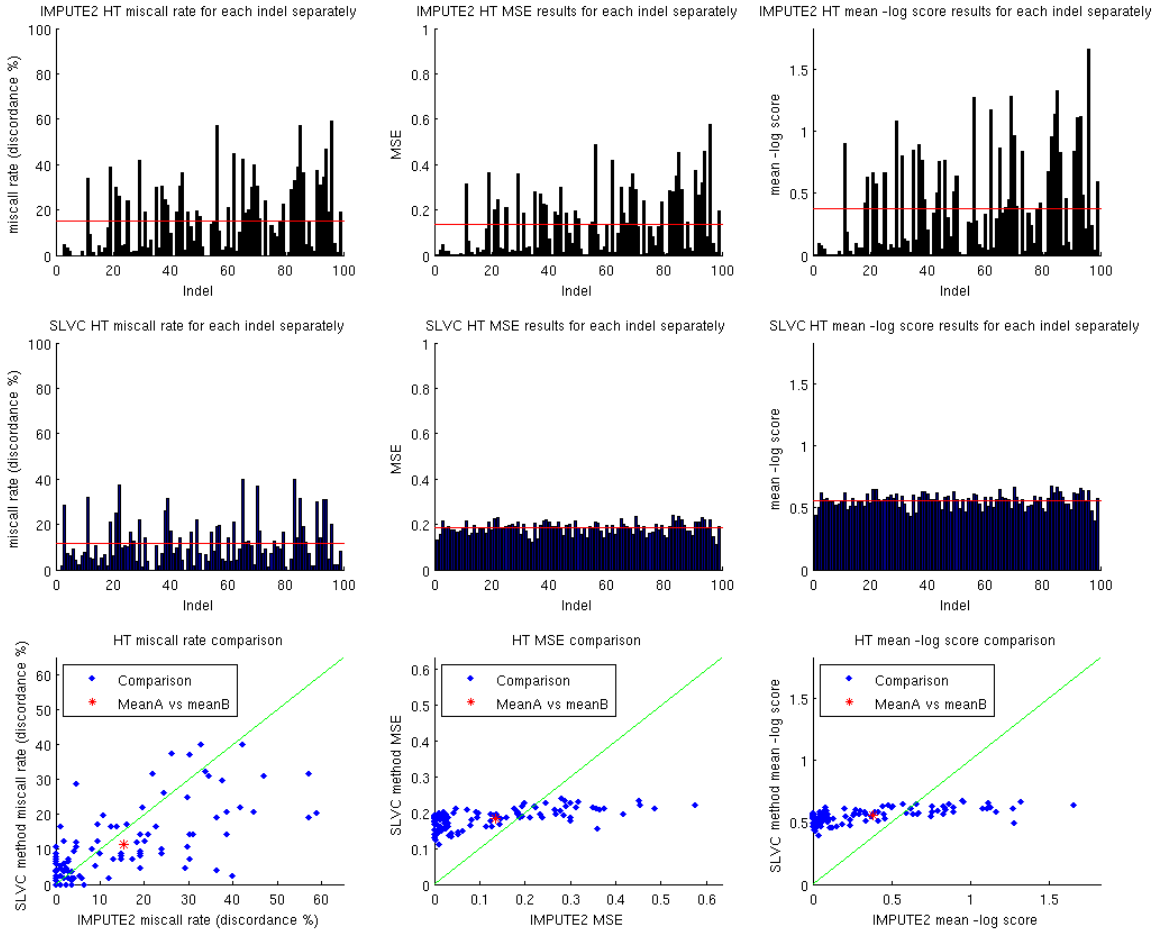

Figure II.17: Summary results using pseudo-indels. The top row presents results from IMPUTE2, the middle row presents SLVC results and the bottom row compares the two sets of results. The first column presents HT miscall rates; the second column presents the mean squared error (MSE) between the true HT probabilities (always 0 or 1) and the predicted probability for the true HTs; the third column presents logarithmic scores with the infinite log scores excluded.

4.3 Results using semi-synthetic pseudo-indel data

A considerable limitation of the validation results in Section 4.1 is that they assessed GT prediction accuracy only. While high GT accuracy is indicative of the method overall validity, further tests were required to assess phasing accuracy. Following the observation that the LD pattern between indels and SNPs is very similar (see Section 2.1), an approximate assessment of the method's phased HT calling accuracy was performed using semi-synthetic pseudo-indels.

4.3.1 Validation and comparisons with IMPUTE2 using pseudo-indel data

SLVC and IMPUTE2 (version 2.2.2) were run using the pseudo-indel set introduced in Section 2.2. IMPUTE2 was run with default settings - an effective population size N_e=20,000, 500kb buffer regions and 100 iterations of which 10 were used as burn-in. SLVC was initialised with the input GTL modes (see Section 3.2.3) and was run for 10,000 iterations, with a threshold t=0.9 and a mutation rate m=0.01. The execution time of the indel calling method was ∼2 hours for similarity matrix production with *lstree* plus ∼11 hours for inference without parallelism. IMPUTE2 took ∼47 hours to run with the same hardware.

Considering that SLVC is expected to converge confidently in less than 1000 iterations, the execution time of SLVC may be expected to run in 1-2 hours, resulting in a considerable minimum 10-12-fold speed difference between the two methods.

Figure II.17 compares SLVC and IMPUTE2 results on the pseudo-indel dataset. The results show that SLVC achieved very similar overall HT call prediction accuracy to IMPUTE2 (11.48% vs 15.28% miscall rate), while the reliability of its HT probability prediction was behind IMPUTE2 (0.1843 MSE vs. 0.1346) according to MSE and mean logarithmic scores. The probability results of SLVC were clustered together around P=0.435, while those of IMPUTE2 were spread out with peaks at 0 and 1 (Figures II.17,II.18). Figure II.18 demonstrates that the clustering of posterior probabilities given by SLVC and the observed disparity between the reliability of calls and the calibration of posterior probabilities comes from the SLVC method being overly conservative and systematically overestimating uncertainty.

Figure II.18 also demonstrates that the relative uncertainty between data points seems incredibly reliable in the case of SLVC, even more so than in IMPUTE2. This observation suggests that it may be possible to apply some transformation to the posterior probability predictions in order to improve their calibration. Figure II.19 demonstrates the effect of applying a linear window type LUT transformation

$$p' = \begin{cases} 1 & \text{when } p \geq WC + WW/2 \\ 0 & \text{when } p \leq WC - WW/2 \\ 2(p - WC)/WW + WC & \text{otherwise} \end{cases} \tag{II.18}$$

which is widely used in signal and image processing, and where p refers to original and p' refers to transformed probabilities. With a window width of WW=0.33 and window center of WC=0.5, the transformed posterior probabilities appear very accurately calibrated. Intuitively, in our context this transformation can be thought of as taking beliefs to more extreme and, hence, potentially countering the conservatism of SLVC.

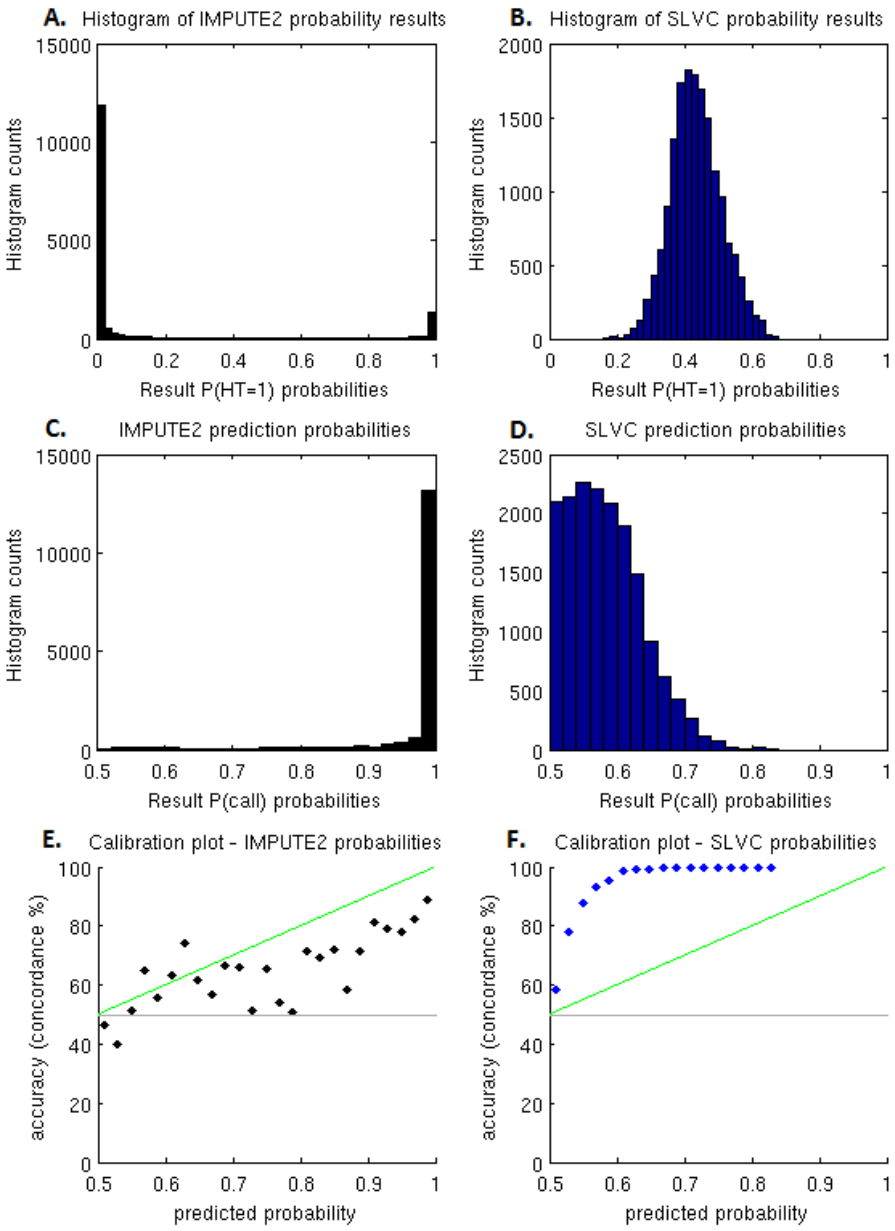

Figure II.18: Summary of posterior probability characteristics. The plots on the left present results for IMPUTE2. The plots on the right present results for SLVC. The histograms on the left illustrate the distribution of HT probability results. The plots on the right show the connection between probability predictions (grouped into 0.02-wide bins) and accuracy (percentage of correct calls) of corresponding results.

A reason to believe in the validity of this transformation and the parameterisation is that the original GTP inputs were strictly in the [0.33, 0.66] interval, likely undermining the 'confidence' of the method, while the transformation worked exactly against that (see Section 2.1.1 for the original data and Section 2.2 for the synthesis of the pseudo-indel data). However, this particular transformation was performed merely for the purpose of better demonstrating that there is some valuable information in the posterior probability results of SLVC. There

is no proven biological or methodological reason for the use of this particular transformation apart from empirical experience with the method, and there is also no proof that it will be well suited for other datasets.

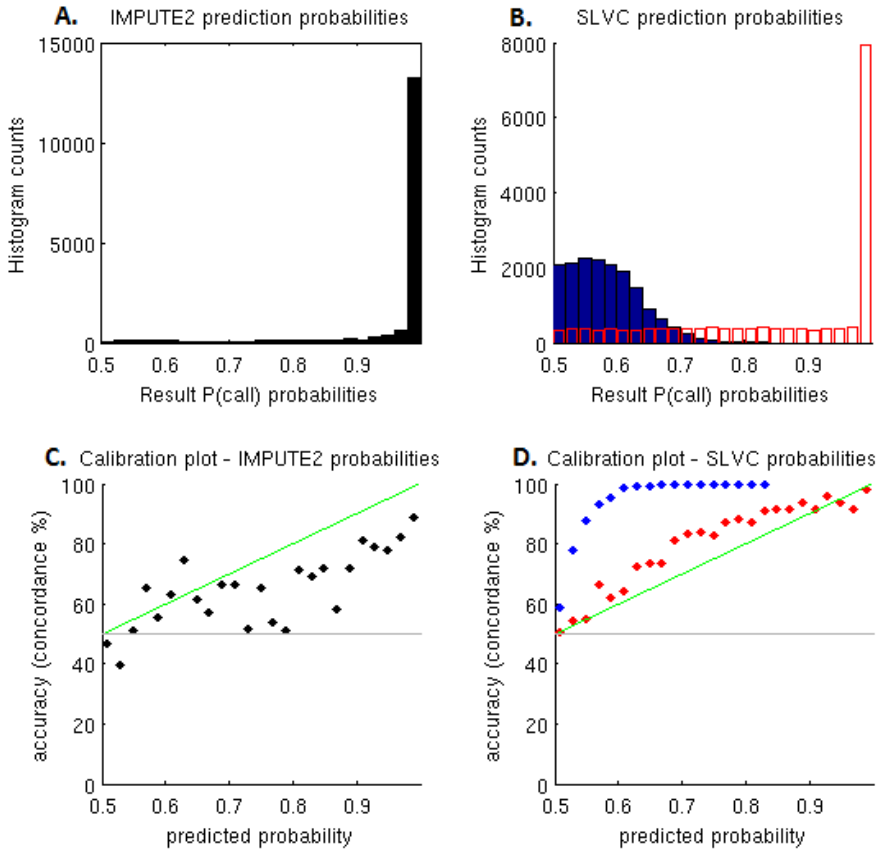

Figure II.19: Correcting calibration with a linear window type LUT transformation. The transformed probability results (red) were acquired by applying a window center WC=0.5 and window width WW=0.33. The results from IMPUTE2 are presented for reference.

Verdict

The results show that the accuracy of SLVC's HT calls is comparable to those of IMPUTE2, which is a one of the most reliable methods in the field.

SLVC also achieves its results faster, with a ×4-10 speedup, even when implemented in MATLAB, compared to IMPUTE2 which was implemented in C. The high potential for parallelisations and optimizsations in both the similarity matrix calculations (see Chapter III) and in SLVC presumes that there may be significantly higher accelerations achievable.

The overly conservative calibration of SLVC's HT probabilities lowers confidence in the

method, but it is likely that effective schemes can be developed to re-calibrate the posterior probability predictions.

5 Conclusions and future work

This chapter presented and discussed a novel single-locus variant caller method, SLVC, designed to make phased haplotype calls based on high-uncertainty genotype likelihood (GTL) data.

The indel data from the early phases of the 1000 Gneomes Project were introduced, including an analysis of the LD structure between SNPs and indels, which served as the basis for development. Next, the SLVC method was presented. SLVC assumes the availability of phased low-uncertainty marker (e.g. SNP) data besides GTLs (for the same individuals), which is used as a scaffold for guiding inference. The scaffold information is used in the form of a similarity measure between individuals, which is then used to guide inference by linking GTL information between individuals.

The accuracy, convergence and calibration characteristics of SLVC were assessed based on results from empirical experiments with a prototype implementation. The results (Section 4) demonstrated that the accuracy of SLVC is competitive (compares well to the highly accurate IMPUTE2 method) despite the simplified single-locus inference approach.

SLVC is also relatively fast converging, coupled with fast practical execution, even with the prototype MATLAB implementation and without parallelisation. Due to its high parallelisability, it is assumed that SLVC may allow for significant speedups compared to alternative methods.

The greatest limitation of the SLVC method is that posterior probabilities are overly conservative. While being conservative may be seen synonymous to being safe, incorrect calibration may feed overly misleading information to downstream applications and, hence, this attribute of the method can not be overlooked.

Future work

It is reasonable to assume that single-locus inference is more tractable than inference with multiple loci due to the reduced search space. The radical increase in the size of genetic datasets (UK10k Human Genome Project, 100,000 Genomes Project, The Million Human Genome Project) may put the SLVC method at an advantage compared to alternative meth-

ods.

Development of the SLVC method was concluded in 2012 and is not expected to continue. However, in case the method was considered for future use because of its favourable performance characteristics, it would be advisable to perform large-scale convergence and accuracy tests to ensure that the model works with large data sets and that the convergence properties also hold. It would also be necessary that the SLVC method is implemented in a high-performance language with adequate optimisations and large-scale parallelisation. Further, if the degree of uncertainty is required by downscale applications, then further development towards the improvement or re-calibration of posterior probabilities may also be required.

Chapter III
PARALLEL METHODS UNDER THE LI AND STEPHENS (LS) MODEL

1 Introduction

Given the significance of the LS model, the importance of exploring the possibilities of parallel implementations can not be doubted. Since the LS model is a Markov model at its core, HMM parallelisation approaches over states and multiple observations provide a good starting point for development.

As mentioned in the introduction (Section I.1.2), when applying the standard HMM algorithms under the LS model, significant modifications have to be made. The simplicity of emissions, the non-homogeneity and the simplified structure and of transition probabilities have to be taken into account, especially if the algorithmic efficiency specific to the LS model is to be achieved. It is therefore essential to develop model-specific parallel implementations of these algorithms to allow for the performance that would be impossible with the use of general parallel implementations.

This chapter presents the application of HMM algorithm parallelisation over the state space coupled with the trivial parallelisation over observations under the LS model. The focus is on parallelisation with GPU architectures in mind, specifically NVIDIA's CUDA platform. Since small changes in algorithm design can mean huge structural changes in parallel implementations, and also to provide a case-study for future development of parallelising statistical methods, the algorithm designs are explained in relative detail.

The practical applicability of parallelising HMM algorithms under the LS model is demonstrated through experimental implementations as well as readily usable applications such as CUDA-Chromopainter (CCP) and a 'likelihood mapper'. Experimental results indicate that significant accelerations are achievable, which can greatly improve and widen the applicability of methods. Large-scale methods can be made more convenient to work with, which would likely improve the quality of research (e.g. through allowing researchers to experiment with different parameterisations and settings). Methods also become applicable using large datasets, which is crucial in population genetics. A potential application area of the parallel methods could be when analysis is time-critical.

Probably the most important purpose of this work is to suggest that the magnitude of acceleration achievable through parallelisation opens up possibilities for re-purposing existing

methods or even developing new methods not possible with sequential and/or low performance implementations.

2 Methods

This section presents the development of parallel HMM algorithms under the LS model with different implementations for experimental and practical application purposes. Although the presented algorithms were specifically aimed for a GPU implementation in CUDA, they may have equal relevance to parallel implementations on other (possibly future) platforms that support efficient communication between parallel threads.

First, an introductory walk-through of parallel algorithm development is given for the Viterbi Algorithm, starting from sequential versions for general HMMs to parallel versions under the LS model (Section 2.1). This walk-through is partially intended to be a guiding example for those who are not experts in parallel programming who wish to attempt implementing statistical algorithms (especially under CUDA).

Next, an analysis is presented on the suitability of the Viterbi algorithm for parallelisation under the LS model, including theoretical run times and time complexities, space requirements and on the limits to potentially achievable execution times and accelerations (Section 2.1.3). A description of a CUDA implementation is given next, including optimisations and limitations (Section 2.1.4).

The parallel Forward-Backward algorithm is described for the LS model in Section 2.2 with theoretical algorithm analyses and descriptions of the corresponding CUDA implementations in a similar but more brief manner, with more focus on practical applicability.

The methods section is concluded with the description of direct applications of parallel HMM algorithms for chromosome painting and large-scale likelihood calculations such as parameter mapping, parameter estimation and dataset (e.g. reference panel) screening.

Notation

The notation used for standard (discrete) HMM algorithms and those under the LS model will be slightly different, as introduced below.

Notation for standard HMMs

- K is the number of observations $\mathcal{K} = \{1, .., K\}$ and $k \in \mathcal{K}$ is the running index over observation sequences. For simplicity, k is often omitted from notation as it is usually convenient to discuss the algorithms with respect to individual observation sequences.
- T (or T_k) is the length of the observation k and $t \in \{1, .., T\}$ is the running index over the positions of the observation sequence k.
- $\boldsymbol{y} = \{y_1, .., y_T\}$ is the observation sequence vector and $y_t = \boldsymbol{y_k}[t]$ denotes the observation at position t.
- $y_{t_m:t_n}$ is the sub-sequence vector $\{y_{t_m}, .., y_{t_n}\}$ between positions t_m and t_n (with $t_n \leq t_m$) and, hence, $y_{1:10}$ refers to the first 10 observations.
- $\mathcal{S} = \{1, .., S\}$ denotes the set of HMM states, while $s_t \in \mathcal{S}$ refers to a state value at position t and $s_{t_m:t_n}$ refers to the state sequence vector $\{s_{t_m}, .., s_{t_n}\}$.
- HMM parameters are defined as $\lambda = (\boldsymbol{E}, \boldsymbol{F}, \boldsymbol{\pi})$. Here the emission probability distributions are denoted by $\boldsymbol{E} = \{E_{jk}\}$, where $E_{jk} = \mathrm{P}(y_t{=}k \,|\, s_t{=}j)$ and $\boldsymbol{F}$ is the transition matrix $\boldsymbol{F} = \{F_{ij}\}$, where $F_{ij} = \mathrm{P}(s_{t+1}{=}j \mid s_t{=}i)$. The initial distribution is given as $\boldsymbol{\pi} = \{\pi_j\}$, where $\pi_j = \mathrm{P}(s_1{=}j)$. In nonhomogeneous HMMs, $\boldsymbol{E}$ and $\boldsymbol{F}$ can vary with t. The parameters λ are sometimes ommitted from the notation.
- $\delta(s_t)$, $\alpha(s_t)$, $\beta(s_t)$ and $\gamma(s_t)$ are recursion terms for the Viterbi pass, the forward and backward passes and the results of the Forward-Backward algorithm, respectively.
- V, A and B will be used to denote important intermediate quantities of forward and backward passes. V^*, A^* and B^* will be used for aggregated (e.g. max or sum) intermediate results.
- In case of analyses and algorithms that involve partitions of the observations, P will denote the number of parts, $p = L/P$ will denote the mean part length (which is the actual part length when all parts are equal in length) and $\boldsymbol{p_i}$ will denote the i^{th} part.

Notation changes under the LS model

- $\mathcal{H}' = \{\boldsymbol{h_1}, .., \boldsymbol{h_K}\}$ is the set of observed haplotypes identified as $\{1, .., K\}$ and indexed by k, which are seen as the observations.
- L (or L_k) is the length of the observation k and $t \in \{1, .., L\}$ is the running index over the loci of the observation sequence k.
- $\boldsymbol{h} = \{h_1, .., h_L\} \in \mathcal{H}'$ is the observation sequence vector and $h_l = \boldsymbol{h_1}[l]$ denotes the observation at locus l.

- $h_{l_m:l_n}$ is the sub-sequence vector $\{h_{l_m}, .., h_{l_n}\}$ between positions l_m and l_n (with $l_n \leq l_m$) and hence $h_{1:10}$ refers to the first 10 loci of the observed sequence h.
- HMM parameters are defined as $\lambda = (\boldsymbol{M}, \boldsymbol{F^{stay}}, \boldsymbol{\pi})$. The emission probability distributions are simply determined by the mutation rates $\boldsymbol{M} = \{m_l\}$, where m_t may be constant across l. The transition probabilities $\boldsymbol{F^{stay}} = \{F_l^{stay}\}$, where $F_l^{stay} = \mathrm{P}(s_{l+1} = s_l \mid s_l)$ are the 'staying probabilities', which also determine the 'switching probabilities' $F_l^{switch} = \mathrm{P}(s_{l+1} \neq s_l \mid s_l) = (1 - F_l^{stay})/N$. The initial distribution is $\boldsymbol{\pi} = \{\pi_j\}$ as before. The parameter λ are sometimes ommitted from the notation.
- $\mathcal{H} = \{\boldsymbol{H_1}, .., \boldsymbol{H_N}\}$ is the set of N reference haplotypes that correspond to HMM states ($\mathcal{H}_k = \mathcal{H}' \setminus \boldsymbol{h_k}$ is possible). Similarly to general HMMs, $s_l \in \{1, .., N\}$ denotes a state at position l and $s_{l_m:l_n}$ refers to the state sequence vector $\{s_{l_m}, .., s_{l_n}\}$.

2.1 An exploratory CUDA implementation of the Viterbi algorithm

The power of HMM algorithms comes from their recursive, dynamic programming nature that allows for efficient computation of exact results. The dynamic programming recursions of HMM algorithms are also particularly suitable for parallelisation, which can make the algorithms many times faster.

Since the Viterbi algorithm is the simplest in terms of achieving numerical stability, it was the first candidate for exploratory parallelisation.

2.1.1 The sequential Viterbi algorithm under the LS model

The standard sequential Viterbi algorithm for general HMMs

The purpose of the Viterbi algorithm is to find the single most probable $\boldsymbol{s^*} = \{s_1^*, .., s_T^*\}$ joint state sequence (also called the Viterbi state sequence, or Viterbi path) corresponding to a given sequence of observations $\boldsymbol{y} = \{y_1, ..., y_T\}$ under given model parameters (Rabiner, 1989).

The Viterbi algorithm is centered around the recursive calculation of the quantity

$$\delta_{s_t} = \max_{s_{1:t-1}} \mathrm{P}(\, s_{1:t},\ y_{1:t} \mid \lambda) \tag{III.1}$$

which denotes the probability of the most likely state sequence $s_{1:t}$ for $y_{1:t}$ that ends in state s_t given the model parameters λ.

The first step of the Viterbi algorithm is to calculate the δ_{s_1} values for each possible starting state s_1 as

$$\delta_{s_1} = \pi(s_1)\,\mathrm{P}(\,y_1 \,|\, s_1\,) \tag{III.2}$$

where $\pi(s_1)$ is the initial probability of state s_1, i.e. the a priori probability of the HMM starting in state s_1 without any information on the sequence of observations. $\mathrm{P}(\,y_1 \,|\, s_1\,)$ is the emission probability of state s_1 corresponding to the first observation y_1.

After the initialisation phase, the Viterbi algorithm proceeds with the calculation of the δ_{s_t} 'Viterbi values' for each possible 'present' state s_t for each piece of observation y_t with the recursion

$$\delta_{s_t} = \mathrm{P}(\,y_t \,|\, s_t\,) \;\; \max_{s_{t-1}} \big\{\, \mathrm{P}(\,s_t \,|\, s_{t-1}\,)\, \delta_{s_{t-1}} \,\big\}. \tag{III.3}$$

where $\mathrm{P}(\,s_t = i \,|\, s_{t-1} = j\,) = F_{ij}$ refers to the transition probabilities and $\mathrm{P}(\,y_t = n \,|\, s_t = j\,) = E_{jn}$ to the emission probabilities.

For notational convenience let us introduce the intermediate quantities

$$V^*_{s_t} = \max_{s_{t-1}} \big\{\, \mathrm{P}(\,s_t \,|\, s_{t-1}\,)\, \delta_{s_{t-1}} \,\big\} \tag{III.4a}$$

$$S^*_{s_t} = \arg\max_{s_{t-1}} \big\{\, \mathrm{P}(\,s_t \,|\, s_{t-1}\,)\, \delta_{s_{t-1}} \,\big\} \tag{III.4b}$$

where $S^*_{s_t}$ can be thought of as the most likely preceding state and $V^*_{s_t}$ can be thought of as the corresponding weighting factor that the most likely preceding path contributes to the calculation of δ_{s_t}.

In the Viterbi algorithm, the most likely preceding states $S^*_{s_t}$ have to be stored as backward pointers for each s_t in an $N{\times}L$ 'backtrack' matrix. Having calculated the Viterbi values and the most likely preceding states from beginning to end, the last stage of the Viterbi algorithm is assembling the Viterbi path. The most likely final state s_T, with the maximum final Viterbi value δ_{s_T} will be saved as the last state s^*_T in the Viterbi path. The preceding state sequence is acquired by recursively following the backward pointers $s^*_{t-1} = S^*_{s^*_t}$ starting from s^*_T. The resulting state sequence is the output of the Viterbi algorithm.

The standard sequential Viterbi algorithm is described under Algorithm 1. From lines 3, 4 and 5, it is clear that the time complexity of the Viterbi algorithm is $\mathcal{O}(TS^2)$ for each sequence.

Algorithm 1 The standard sequential Viterbi algorithm for general HMMs

init: Calculate the δ_{s_1} values for all $s_1 \in \{1, ..., N\}$ according to eq. (III.2);
for all observation sequences $\{1, ..., K\}$ **do**
  for all pieces of observation $t \in \{2, ..., T\}$ **do**
    for all possible 'present' states $s_t \in \{1, ..., S\}$ **do**
      Find $V^*_{s_t}$ by going over all possible preceding states $s_{t-1} \in \{1, ..., S\}$;
      Save $S^*_{s_t}$ corresponding to $V^*_{s_t}$ as a backtrack pointer from s_l;
      Calculate the δ_{s_t} Viterbi value according to eq. (III.3);
    end for
  end for
end for
Find $\arg\max_{s_T} \delta_{s_T}$ and store it as s^*_T;
backtrack: Trace pointers backwards from s^*_T to get the Viterbi path $\{s^*_1, ..., s^*_T\}$.

Modifications under the LS model

A sequential implementation of the Viterbi algorithm under the LS model mostly follows the general Viterbi algorithm. However, because of the simplified transition structure in the LS model, it is possible to achieve a factor of $\mathcal{O}(N)$ algorithmic acceleration with model-specific implementations as demonstrated in (Li and Stephens, 2003) for the forward algorithm.

The notation used in connection with the LS model follows the one introduced in Sections I.1.2 and III.2. Accordingly, the observation sequences $\boldsymbol{y}$ are replaced by 'observed haplotypes' $\boldsymbol{h}$ defined over a set of $l \in \{1, ..., L\}$ loci. The HMM states $s \in \{1, ..., N\}$ each correspond to one of the $H_s \in \{H_1, ..., H_N\}$ reference haplotypes (where a 'self' reference set $\mathcal{H}_k = \mathcal{H}' \setminus \boldsymbol{h_k}$ is allowed). The model's transition and emission probabilities are defined as in Section I.1.2, by equations (I.1) and (1.2).

For simplicity, let us define the intermediate quantities

$$V_l^{*switch} = \max_{s_{l-1}} \left\{ \mathrm{P}(s_l \mid s_{l-1})\, \delta_{s_{l-1}} \right\} = \max_{s_{l-1}} \left\{ F_{l-1}^{switch}\, \delta_{s_{l-1}} \right\} \tag{III.5a}$$

$$V_{s_l}^{stay} = \mathrm{P}(s_l = s_{l-1} \mid s_{l-1})\, \delta_{s_{l-1}=s_l} = \left(F_{l-1}^{stay} + F_{l-1}^{switch} \right) \delta_{s_{l-1}=s_l} \tag{III.5b}$$

$$V^*_{s_l} = \max_{s_{l-1}} \left\{ \mathrm{P}(s_l \mid s_{l-1})\, \delta_{s_{l-1}} \right\} = \max \left\{ V_l^{*switch}, V_{s_l}^{stay} \right\} \tag{III.5c}$$

and

$$S^*_{s_l} = \arg\max_{s_{l-1}} \left\{ \mathrm{P}(s_l \mid s_{l-1})\, \delta_{s_{l-1}} \right\} \tag{III.5d}$$

where

$$F_l^{stay} = e^{-\rho_l} \tag{III.5e}$$

$$F_l^{switch} = \frac{1 - e^{-\rho_l}}{N} \tag{III.5f}$$

and both are independent of s_l but can vary across loci and the formulas come from the $\mathrm{P}(\, s_l \,|\, s_{l-1} \,)$ transition probabilities of the LS model (see Section I.1.2). As before, $S^*_{s_l}$ is the most likely preceding state and $V^*_{s_l}$ is the corresponding weighting factor.

Algorithm 2 A sequential Viterbi algorithm for the LS model

init: Calculate the δ_{s_1}, $V^{stay}_{s_1}$ and $V^{*switch}_1$ values for all $s_1 \in \{1, ..., N\}$;
for all inference haplotypes $h_1, ..., h_K$ **do**
  for all loci $l \in \{2, ..., L\}$ **do**
    Initialise $V^{*switch}_{l+1}$ as 0;
    for all possible 'present' states $s_l \in \{1, ..., N\}$ corresponding to $H_1, ..., H_N$ **do**
      Calculate $V^*_{s_l} = \max\{\, V^{stay}_{s_l},\ V^{*switch}_l \,\}$;
      Save $S^*_{s_l}$ corresponding to $V^*_{s_l}$ as a backtrack pointer from s_l;
      Calculate the δ_{s_l} Viterbi value with $V^*_{s_l}$ according to eq. (III.3);
      Pre-calculate $V^{stay}_{s_{l+1}}$ according to eq. (III.5c);
      Update $V^{*switch}_{l+1} = \max\{\, V^{*switch}_{l+1},\ F^{switch}_l\, \delta_{s_l} \,\}$;
    end for
    Save the final $V^{*switch}_{l+1}$ for use in iteration $l+1$;
  end for
end for
Find $\arg\max_{s_L} \delta_{s_L}$ and store it as s^*_L;
backtrack: Trace pointers backwards from s^*_L to get the Viterbi path $\{s^*_1, ..., s^*_L\}$.

Algorithm 2 presents a simplistic outline of the sequential Viterbi algorithm modified for the LS model. The $\mathcal{O}(N)$ algorithmic speedup is achieved by pre-calculating $V^{stay}_{s_{l+1}}$ and $V^{*switch}_l$ in the third loop instead of having a fourth inner loop.

2.1.2 Parallel algorithm design

The first step when developing parallel algorithms is to identify the operations that can be performed simultaneously, i.e. uncovering the best possibilities for parallelism. For instance, the initialisation loops can be trivially parallelised but in most cases this will not result in any significant speedup. The focus here is the parallelisation of the nested loops of the main recursion over both the observations and the state space. In standard HMM algorithms it is also possible to parallelise the state space-wide reduction operations (summations and maximisations), but under the LS model these summations need to be merged into the parallelised loop over the 'present' states (see in the following explanation). Since the parallelisation of reductions is more a necessary by-product than the focus of this work, the corresponding details will be omitted from the derivation of the parallel algorithms for simplicity.

As mentioned in Section I.4, parallelisation of the outer loop over observations is trivial. A

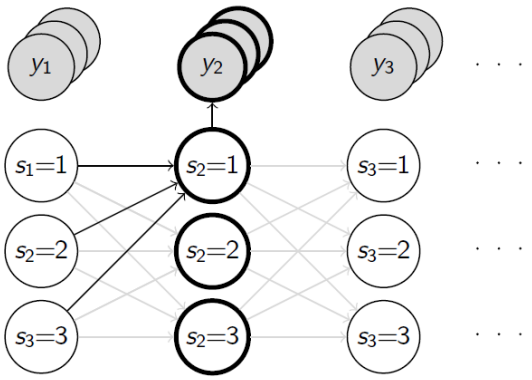

Figure III.1: Parallelisation of HMM algorithms over the observation set and over the state space (the 'present' states). Computation for the bold circles may be performed in parallel. The arrows present message passing. Reduction (maximisation or summation) has to be performed over messages that go into the same node ('present state').

more interesting parallelisation can be achieved over the states s_l for calculating the δ_{s_l} values simultaneously. As explained in the previous section, the only way for any implementation under the LS model to reach the highest potential algorithmic efficiency is by pre-calculating the $V_{l+1}^{*switch}$ values.

Unfortunately in a parallel setting maximisation can only be achieved by exchanging information between parallel threads. Since parallel threads are not guaranteed to execute or finish in any particular order, care has to be taken to maintain the correctness of calculations and achieve performance in the same time by appropriate synchronisation. The synchronised communication between threads can be achieved efficiently by a suitable parallelised 'reduction' algorithm (see Figure I.10 in Section I.3.1.3 for a diagrammatic explanation). Assuming that each parallel thread is set to carry out the computation of δ_{s_l} for a different 'present' state s_l, the required maximisation reduction is performed as follows. Each one of the N threads starts with a value $V_{s_{l+1}}^{switch} = \mathrm{P}(s_{l+1} \mid s_l)\, \delta_{s_l}$. In a loop of $log(N)$ iterations, the threads are paired according to a binary tree and only the higher value from each pair is passed to the next maximisation iteration, halving the number of 'active' threads in each step until the final single active thread obtains the global maximum value.

Figure III.1 demonstrates the parallelisation over observations and 'present states'. Algorithm 3 presents a version of the Viterbi algorithm for the LS model that is parallelised both over the set of inference haplotypes and the 'present' states.

Algorithm 3 Pseudocode for a parallel Viterbi algorithm for the LS model

init: Calculate the δ_{s_1}, $V_{s_1}^{stay}$ and $V_1^{*switch}$ values for all $s_1 \in \{1, ..., N\}$;
parallel for all observed haplotypes $h_1, ..., h_K$ **do**
  for all loci $l \in \{2, ..., L\}$ **do**
    parallel for all 'present' states $s_l \in \{1, ..., N\}$ corresp. to $H_1, ..., H_N$ **do**
      Calculate $V_{s_l}^* = \max\{ V_{s_l}^{stay}, V_l^{*switch} \}$;
      Save $S_{s_l}^*$ corresponding to $V_{s_l}^*$ as a backtrack pointer from s_l;
      Calculate the δ_{s_l} Viterbi value with $V_{s_l}^*$ according to eq. (III.3);
      Calculate $V_{s_{l+1}}^{stay}$ according to eq. (III.5c);
      Calculate $\mathrm{P}(s_{l+1} \,|\, s_l)\, \delta_{s_l}$;
    end parallel
    Initialise the $V_{l+1}^{*switch}$ with $V_{s_{l+1}}^{switch}$ in each thread;
    Synchronise threads working with the same h_k haplotype;
    Find $V_{l+1}^{*switch}$ with parallel reduction and save for use in iteration $l+1$;
  end for
end parallel
Find $\arg\max_{s_L} \delta_{s_L}$ and store it as s_L^*;
backtrack: Trace pointers backwards from s_L^* to get the Viterbi path $\{s_1^*, ..., s_L^*\}$.

2.1.3 Algorithm analysis and considerations for implementation

Suitability for parallelisation on CPUs and GPUs

As discussed in Section I.4.1, the trivial parallelisation of HMM algorithms over observations can be achieved by GPUs, multi-core CPUs, CPU clusters and distributed computing architectures alike.

Parallelisation over the state-space, however, requires the facility of efficient communication between parallel threads. For this reason, GPUs are currently the only suitable hardware platform for this particular parallelisation. For instance, CUDA GPUs allow communication between threads of a block to be performed with parallel access to relatively fast shared memory (SMEM).

Since GPUs uniquely allow for both types of parallelisation, all consequent analyses and results are presented with a focus on GPU implementations.

Space requirements and data intensity

The most important restrictions and complications for achieving the highest possible performance with GPUs are related to data movements and memory usage. GPUs have limited

memory space, high memory access latencies and high (but still limited) memory bandwidths. GPUs are also separated from host resources, which implies that data transfers are required between host and device. These transfers also have communication costs (latency and transfer bandwidth) and consequently, GPUs have strong parallelisation and efficiency constraints for memory intensive applications (Bauer et al., 2011; Bordawekar et al., 2010; Kirk and Hwu, 2010; NVIDIA, 2014).

By their nature, HMM algorithms require large amounts of data and are memory-bound, meaning that performance is mostly limited by memory operations and data transfers rather than calculations. Since the purpose of parallelisation is practical performance, it is fundamental to analyse memory requirements.

The Viterbi algorithm requires large pieces of data to be readily available in device (meaning GPU) memory space (GMEM). The LS model introduces both lowered and increased memory requirements at the same time, when compared to standard HMM algorithms. The LS model allows for the transition probabilities to change across loci based on local recombination rates, but storage of a single F_l^{stay} value for each l is sufficient (as a $(1{\times}L)$ sized vector). It is also helpful that under the LS model all information on the emission probabilities can be represented as a single mutation rate for the whole recursion. The first significant memory requirements are those of the $\mathcal{H}'$ and $\mathcal{H}$ haplotypes and the temporary results $V_l^{*switch}$, S_l^{*jump} and $V_{s_l}^{stay}$ using $\big(K{\times}L + N{\times}L + 3{\times}K{\times}N\big)$ space. The real difficulty, however, is that the backtrace matrices that include all $S_{s_l}^*$ need to be stored for each $\boldsymbol{h}{\in}\mathcal{H}'$, requiring a total memory space of $K{\times}L{\times}N$ bytes (see Table III.1 for examples).

Table III.1: Summary table of space requirements of the parallel Viterbi algorithm for the LS model

Data size			Required device memory (MB)		
K	L	N	Reference hap. data	Inference hap. data	Backtrace matrix
32000	7500	2	∼0	229	894
2000	7500	64	0.5	14	1788
2000	4000	128	0.5	8	1907
2000	2000	256	0.5	4	1907
500	2000	1024	2	1	1907
1000	1000	1024	1	1	1907
2000	500	1024	0.5	1	1907

$K = |\mathcal{H}'|$ is the number of inference sequences, L is the number of loci (the length of inference and reference sequences) and $N = |\mathcal{H}|$ is the number of reference sequences

Pre-requisites for performance

In order to reach the acceleration ceiling with GPUs, it is essential that variables and data are always stored in the most suitable fast-access memory spaces, such as the constant memory (CMEM), the L1 and L2 caches, the texture memory and the shared memory (SMEM) (see Section I.3.1 and Table I.1 for details). Although these memory spaces are much smaller (in the range of 16-48kBs) than the global device memory (GMEM, 2-12GB on current GPUs) and can be complicated to manage, their advantageous access properties are crucial for achieving efficiency (Kirk and Hwu, 2010; Nickolls et al., 2008; Owens et al., 2008; Ryoo et al., 2008).

Time complexities and execution times

Theoretical analysis

It is important to note that in the case of parallel algorithms the number of operations and the actual computation times do not usually align in the way it is assumed in sequential algorithms. In fact the objective of parallelisation itself is to reduce the execution time of an algorithm of given complexity. Consequently, in the case of parallel algorithms the terms 'computational complexity' and 'execution time' are better suited and will be used instead of time complexity.

For the sake of theoretical analysis, all algorithms are assumed to be run on large, arbitrary datasets with $|\mathcal{H}'|=K$ inference sequences (or observations) and $|\mathcal{H}|=N$ (or $N=|\mathcal{S}|=S$) reference haplotypes (or states) defined on the same arbitrary set of $L(=T)$ loci (or time steps).

The time complexity of a sequential Viterbi algorithm implemented for general HMMs is $\mathcal{O}(KLN^2)$. This time complexity can be reduced algorithmically to $\mathcal{O}(KLN)$ for the LS model. With the two-way parallelisation over $\mathcal{H}$ and $\mathcal{H}'$ presented in Section 2.1.2, the theoretically achievable minimum execution time is $\mathcal{O}(L \log N)$ under the LS model. It is not possible to achieve an N factor acceleration (for single sequence runs) because the parallelisation over N states requires that the $F_{s_{l+1}}^{switch}$ intermediate results are 'reduced' over all states (the maximum of results from different threads needs to be found) at each locus (see Algorithm 3). This reduction operation can not be moved into any of the higher loops, and even with a tree-based parallel reduction, the calculation takes an order of $\mathcal{O}(\log N)$ time. Accordingly, the parallelisation over N states only results in an $\mathcal{O}(N/\log N)$ theoretical speed increase. Moreover, due to the necessity of performing the reduction, the algorithm requires $\mathcal{O}(KLN \log N)$ operations to achieve its $\mathcal{O}(L \log N)$ execution time with a $K \times N$-fold parallelisation. The $\mathcal{O}(\log N)$ increase in theoretical computational complexity is the

price that has to be traded for the lower execution time achievable through parallelisation in the case of the LS model.

Clearly, the trivial parallelisation over $\mathcal{H}'$ could be easily achieved outside the realm of GPUs and without much effort in parallel programming (as discussed in Section I.4.1). However, because GPUs can perform better when doing multiple levels of parallelisation, it is beneficial to include this parallelisation along that over $\mathcal{H}$. Inclusion of parallelisation over $\mathcal{H}'$ allows for the CUDA scheduler to hide the latency of memory operations with greater amounts of computation.

In practice

It has to be emphasised that the above arguments are purely theoretical. In practice the actual number of operations and the implementation details can make a significant difference for the final performance. For instance, the theoretical comparisons omitted the fact that all the operations that the maximum calculation requires in the parallel algorithm are necessarily present in the sequential algorithms as well. Those calculations are just 'hidden' in one of the loops that is required regardless of the max calculation - they do not affect the order in which the execution time of the algorithm increases with the increase of the data size, but they affect the actual execution time.

Naturally, in practice there exists an effective practical parallel execution limit based on hardware resources, beyond which execution is serialised by the hardware. Due to the specialised GPU architecture, GPUs are able to run 4-12 blocks on each SM, with 4 warps running in each block, as long as the number of threads and blocks is sufficient. The result is about 32-48 warps and 1024-1536 threads running at the same time per SM. With multiple SMs (2-15), the maximum concurrent execution achievable is about 2-23k. Most of the highly effective parallelism is a consequence of the scheduling/pipelining capabilities that allow CUDA GPUs to parallelise memory access, which would otherwise dominate computation.

Another important practical restriction in CUDA is that the maximum number of threads allowed is only 1024 per thread block (under current versions of CUDA), however, it is possible to overcome this restriction by 'paging' virtual threads (see Section I.3.1.3 for a general explanation, or later sections for a specific implementation example for the forward-backward algorithm). Further, GPUs are generally slower in terms of clock frequency and have a lower single-thread efficiency than CPUs. Accordingly, when comparing efficient GPU implementations to efficient sequential CPU implementations, it translates into the overall acceleration achieved.

The final speedup comes from the interplay of multiple factors. In practice, when parallel GPU applications were compared to optimised sequential CPU applications, accelerations

have been observed to range from roughly ×4 to over ×200 (Nickolls and Dally, 2010; Ryoo et al., 2008; Zhang et al., 2009).

When compared to multi-threaded parallel CPU implementations, it is generally believed that GPUs are currently able to provide a maximum of 20 to 30-fold and an average of 2 to 10-fold acceleration. In some cases, the speed of parallel CPUs may be able to match GPUs (Bordawekar et al., 2010; Lee et al., 2010; Nickolls and Dally, 2010; Owens et al., 2008; Vuduc et al., 2010). The Viterbi algorithm under the LS model is relatively suitable for GPUs, despite the inevitable $\log N$ cost of parallel reductions.

2.1.4 Implementations

Sequential programming usually does not require extensive optimisation efforts from the programmer in order to achieve acceptable performance because most sequential compilers are highly capable of automatic optimisations.

Unfortunately, the case is rather different in parallel programming. The performance of parallel implementations is highly dependent on the specific implementation details such as the optimisations applied. The parallel execution of a number of threads is usually not sufficient on its own. Without the appropriate use of GPU resources, capabilities and optimisations a CUDA implementation may well be slower than a sequential implementation. Without considering the specific limitations and mechanisms of GPU architectures, threads may execute effectively sequentially. Moreover, CUDA compilers do not perform automatic optimisations to a great extent and are, naturally, incapable of performing optimisations on a structural level. In possession of domain-level knowledge, it is the responsibility of developers to devise most optimizations, which are an integral part of CUDA programming.

The parallel Viterbi algorithm was implemented for the purpose of serving as a proof of concept for developing further parallel HMM algorithms and practical applications under the LS model. The implementations presented here demonstrate the efficient use of most typical CUDA capabilities.

Sequential MATLAB, sequential C++ and parallel CUDA versions of the Viterbi algorithm have been implemented for empirical efficiency and accuracy comparisons. All implementations assumed the LS model as explained in Section 1.2, with an additional restriction that $\mathcal{H}' \cap \mathcal{H} = \emptyset$. The sequential C++ and the CUDA implementations use single precision floating point numbers to achieve performance while MATLAB uses double precision floating point numbers. All calculations are carried out in log space for numerical stability. The sequential C++ and CUDA codes were compiled with optimization level `O1` and without the use of 'fast math' operations, because this choice proved to help retain accuracy.

Description of implementations and optimisations

Both the sequential MATLAB and C++ implementations were based on Algorithm 2 (Section 2.1.1). The sequential MATLAB version was used only for prototyping and as an accuracy benchmark and, hence, it was not optimised. The sequential C++ implementation served both as an implementation starting point and as an efficiency benchmark for the parallel CUDA development. The C++ code is simple and moderately optimised and was deliberately kept very similar to the CUDA code.

Due to the complexity of device-specific parallel programming under CUDA, the parallel implementation was achieved by defensive programming (which means that extensive testing was performed after each small incremental step in the development process). The final CUDA implementation is based on Algorithm 3 (Section 2.1.2), achieving the two-way parallelism over the set of inference sequences (observations) and the model states (with an internal loop over loci). The final kernel is designed to run with K thread blocks and $N \leq 1024$ threads - one thread block for each inference sequence and one thread for each reference sequence (each possible state). Each block k performs an independent run on one of the K given inference haplotypes $\boldsymbol{h_k} \in \mathcal{H}'$, while the threads correspond to the N model states.

Since the performance of parallel applications often depends on the balance of data transfers, memory operations and calculations and the Viterbi algorithm was expected to be memory-intensive, memory management was a key area of focus. Each kernel run requires the staying and switching probabilities for each locus as well as the reference and inference sets for the set of loci it is to be run on. Kernel runs also need to return the backtrack matrix they produce.

Scalar parameters (e.g. N, K, L), mutation rates, and staying and switching probabilities are all directly transferred to and are accessed from CMEM for optimisation purposes. Because all threads in a warp can be expected to perform calculations using the same staying and jumping probabilities, the kernels achieve a speed boost by exploiting the broadcast capability of the constant memory. In addition, the locality of consecutively accessed staying and jumping probability values (all running blocks and threads can be expected to perform calculations around the same loci) automatically allows for optimal use of the CMEM cache. Because calculations are in log space to ensure numerical stability and the calculation of logarithms is inefficient on the GPU, it was better to minimise the number of required log operations. Hence, precursor log values for transition probabilities are stored as a single vector of $\exp(-\rho_l \, d_l \, / N \,)$ values.

The Viterbi kernel requires both the $\mathcal{H}'$ and $\mathcal{H}$ haplotype sets to be available. These pieces of data are first transferred into GMEM, then each $\boldsymbol{h_k}$ is copied (in parallel) into the SMEM

of each thread block. Each block is required to access only the associated $\boldsymbol{h_k}$, not the whole set of $\mathcal{H}'$. The intermediate calculation quantities ($V_l^{*switch}$ and $V_{s_l}^{stay}$) were also chosen to be stored in SMEM to minimise the use of registers and the access to slow global memory. The only two pieces of data that are kept in and accessed from GMEM are the reference data ($\mathcal{H}$) and the backtrack matrix ($S*_{s_l}$).

The memory layouts are designed such that array elements that are expected to be accessed at the same time are placed in close proximity to each other (this is called coalesced memory layout). Coalesced memory layout allows parallelised access of SMEM in a few cycles and supports efficient caching for the GMEM and CMEM.

It would have been possible to move the reference dataset ($\mathcal{H}$) to constant memory, but that would have further limited the number of loci with which the kernel can run and hence this option was discarded. It could also have been possible to move the reference dataset into SMEM, but then each block would have needed to have a copy of all $\mathcal{H}$ haplotypes, making SMEM use very inefficient. A potential improvement could have been the use of the texture memory for storing and efficiently accessing the reference haplotypes, but this option has not been investigated due to the complexities and particularities of texture memory handling.

The most critical step in the algorithm is the maximisation reduction step. It was separated into two stages to minimise the number of required synchronisations that could lower efficiency. Inter-warp reduction requires synchronisation at each reduction iteration, while intra-warp reductions (the last 5-6 reduction iterations) are inherently synchronised due to the architectural arrangement that all threads in a warp are executed at the same time.

To further improve efficiency, branching (the number of if-then-else statements) was minimised and operations (such as calculations) were performed with the fastest available equivalent options (e.g. bit shifts were used instead of integer divisions).

In CUDA the number of threads can be limited by the number of required registers. Hence, special care was taken when structuring the calculation and memory excess pipelines of the Viterbi algorithm to allow for the maximum possible number of threads (and hence maximum parallelism) by keeping the number of required registers to be minimum. The absolute highest number of threads running in parallel (1536) could only be achieved using less than 22 registers per thread (for Fermi GPUs). Unfortunately, the final parallel implementation uses exactly 22 registers. Lowering the number of registers through reorganisation of memory access and computation order can sometimes improve efficiency, but can sometimes result in the introduction of serialised executions and data delays. All efforts towards lowering the number of required registers below 22 have resulted in longer measured execution times and, hence, development was concluded with the use of 22 registers.

Limitations and future work

Due to the deliberate simplicity of the design, states ('present' states) are directly assigned to threads. Therefore, the number of maximum allowed CUDA threads in each thread block (1,024 on current CUDA architectures) limit the allowed datasets to have $N \leq 1024$. This limitation can be overcome in future development by paging virtual threads (as demonstrated for the Forward and Forward-Backward algorithms in the following Section 2.2).

The size of the global memory means a hard constraint on the size of the backtrack matrix and, thus, poses a hard constraint on the possible configurations of K and N together. Assuming that the backtrack matrix is stored as a short integer array, the largest data sizes are defined by the constraint $c \times K \times L_p \times N < \text{Size}_{\text{GMEM}}$, where c is the size of the data type 'short' in bytes (usually 2 bytes). GMEM sizes characteristically fall in the range of 2-12GBs. when implemented for practical use, the GMEM limitation can be countered by CUDA streaming - as explained and demonstrated for the Forward and Forward-Backward algorithms in the following section (Section 2.2). In the absence of streaming, problems may be divided over subsets of the $\mathcal{H}'$ inference sequences without altering results.

The size of CMEM (64kB) limits the length of the staying and jumping probability vectors to be around 7500 irrespective of the configuration of K and N (NVIDIA, 2014). This limitation is only relevant in the absence of streaming.

The requirement of 22 registers possibly poses a limit on the number of maximum threads and hence possibly indirectly limits the flexibility of the CUDA thread block scheduler. Both restrictions may have a slight negative effect on performance depending on the exact data size.

In CUDA, any code development with the aim of optimisation is volatile. Minor changes in design often require major restructuring of the code, memory allocations, memory layouts and operation pipelines (Owens et al., 2008). The current CUDA code is optimised to a great extent, but most likely not to the absolute limit. Texture memory has not been used in the implementations. Harnessing texture memory for storing reference data could be one option for future exploration. The use of scaling instead of log space and hence the storage of a single vector in the constant memory might also lead to some improvements.

Haplotype alleles are currently stored as short integers. Although the strongest memory and efficiency constraints are not directly related to the haplotype data sets themselves, the encoding of haplotypes as individual bits of *int* elements is a possibility for reducing their size to about 1/8 for the price of some extra decoding operations.

2.2 The Forward and the Forward-Backward algorithms

The Forward and Forward-Backward (FB) algorithms are frequently used in genetics and were the natural choice for practical parallel implementation under the LS model. The version of the Forward algorithm that only stores results for the last locus will be called the standalone Forward (SF) algorithm as it can be used without a backward pass. Omitting the explanation of the general SF and FB algorithms, the sequential versions under the LS model are introduced directly.

2.2.1 Parallel algorithm design

The sequential SF and FB algorithms under the LS model

The Forward algorithm

The Forward algorithm computes the joint probabilities $\alpha_{s_l} = \mathrm{P}(h_{1:l}, s_l)$ with a single forward pass through loci $\{1, .., L\}$ according to the recursion

$$\alpha_{s_l} = \mathrm{P}(\, h_l \,|\, s_l \,) \sum_{s_{l-1}} \Big(\mathrm{P}(\, s_l \,|\, s_{l-1} \,)\, \alpha_{s_{l-1}} \Big) \tag{III.6a}$$

$$\alpha_{s_1} = \pi(s_1)\, \mathrm{P}(\, h_1 \,|\, s_1 \,). \tag{III.6b}$$

At the end of the Forward algorithm, one can compute the probability of a sequence $\mathrm{P}(h_{1:L})$ (or $\mathrm{P}(h_{1:L} \,|\, \lambda)$ to be explicit) or the probability $\mathrm{P}(s_L \,|\, h_{1:L})$ of finishing in a given state s_L as

$$\mathrm{P}(h_{1:L}) = \sum_{s_L} \mathrm{P}(h_{1:L}, s_L) = \sum_{s_L} \alpha_{s_L} \tag{III.7a}$$

$$\mathrm{P}(s_L \mid h_{1:L}) = \frac{\mathrm{P}(h_{1:L}, s_L)}{\mathrm{P}(h_{1:L})} = \frac{\alpha_{s_L}}{\sum\limits_{s_L} \alpha_{s_L}}, \tag{III.7b}$$

where the latter simply means the normalisation of α_{s_L} values.

It is clear that the recursion steps of the Forward algorithm work very similarly to the recursion steps of the Viterbi algorithm. The main differences are that summations are used instead of maximisations, the results are stored as a matrix of probabilities instead of a backtrace matrix and calculations can not be performed in log space. In order to achieve numerical stability, the α_{s_L} have to be kept in a stable range by normalising them at each iteration. The normalising constants are saved as scaling factors for later use.

The FB algorithm

The FB algorithm is designed to find the marginal state probabilities $\mathrm{P}(s_l \mid h_{1:L}, \lambda)$ across $\mathcal{S}$ for all $l \in \{1, .., L\}$, given a complete sequence of observations. The FB algorithm calculates the probability of each possible state at each locus as

$$\gamma_{s_l} = \mathrm{P}(\, s_l \mid h_{1:L}, \lambda). \tag{III.8}$$

The difference from the Forward algorithm is that the information from succeeding observations ($h_{l+1:L}$) are also accounted for.

The γ_{s_l} probabilities are calculated as a product of forward and backward values α_{s_l} and $\beta_{s_l} = \mathrm{P}(h_{l+1:L}, s_l)$ as

$$\gamma_{s_l} = \frac{\alpha_{s_l}\beta_{s_l}}{\mathrm{P}(h_{1:L} \mid \lambda)} = \frac{\alpha_{s_l}\beta_{s_l}}{\sum\limits_{s_l}^{N} \alpha_{s_l}\beta_{s_l}}, \tag{III.9}$$

where $\mathrm{P}(h_{1:L} \mid \lambda) = \sum\limits_{s_l}^{N} \alpha_{s_l}\beta_{s_l}$ is a normalising factor.

The FB algorithm is usually performed by first running a Forward algorithm (or forward pass) to obtain the α_{s_l} values and then running a backward pass to obtain the β_{s_l} values and the γ_{s_l} results.

The initialisation step of the backward pass of the FB algorithm is simply setting $\beta_{s_L} = 1$. The recursion is

$$\beta_{s_l} = \sum_{s_{l+1}} \Big(\mathrm{P}(\, h_{l+1} \,|\, s_{l+1}\,)\ \ \mathrm{P}(\, s_{l+1} \,|\, s_l\,)\, \beta_{s_{l+1}} \Big). \tag{III.10}$$

Normalisation (and the saving of scaling factors) is necessary for both the forward and backward passes of the FB algorithm's numerical stability.

Algorithmic explanation

For notational convenience, let us introduce the following terms for the forward pass

$$\begin{aligned}
A^{switch}_{s_{l-1}} &= \mathrm{P}(\, s_l \,|\, s_{l-1}\,)\, \alpha_{s_{l-1} \neq s_l} \;=\; F^{switch}_{l-1}\, \alpha_{s_{l-1} \neq s_l} \\
A^{*switch}_{l} &= \sum_{s_{l-1} \neq s_l} \Big(F^{switch}_{l-1}\, \alpha_{s_{l-1}} \Big) \\
A^{stay}_{s_l} &= \; F^{stay}_{l-1}\, \alpha_{s_{l-1} = s_l} \\
A^{*}_{s_l} &= \; \sum_{s_{l-1}} \Big(\mathrm{P}(\, s_l \,|\, s_{l-1}\,)\, \alpha_{s_{l-1}} \Big) = A^{*switch}_{l} + A^{stay}_{s_l}
\end{aligned}$$

where $F_{l-1}^{switch}=(1-e^{-\rho_l})/N$ and $F_{l-1}^{stay}=e^{-\rho_l}$ similarly to the Viterbi algorithm, with

$$\alpha_{s_l} = \mathrm{P}(\, h_l \,|\, s_l \,)\ A^*_{s_l}.$$

It is worth noting that due to the normalisation of the α_{s_l} at each iteration,

$$A^{*switch}_{s_l} = F^{switch}_{l-1} \tag{III.12}$$

can be used in practice to save memory operations, space and calculations.

Similarly to the backward pass, let

$$\begin{aligned}
B^{switch}_{s_{l+1}} &= \mathrm{P}(\, h_{l+1} \,|\, s_{l+1} \,)\, \mathrm{P}(\, s_{l+1} \,|\, s_l \,)\, \beta_{s_{l+1}} \quad = \quad \mathrm{P}(\, h_{l+1} \,|\, s_{l+1} \,)\, F^{switch}_l\, \beta_{s_{l+1}} \\
B^{*switch}_l &= \sum_{s_{l+1}} B^{switch}_{s_{l+1}} \\
B^{stay}_{s_l} &= \mathrm{P}(\, h_{l+1} \,|\, s_{l+1} \,)\, F^{stay}_l\, \beta_{s_{l+1}}
\end{aligned}$$

where $\mathrm{P}(\, h_{l+1} \,|\, s_{l+1} \,)$ is the emission probability of state s_{l+1} at locus $l+1$ and

$$\beta_{s_l} = B^{stay}_{s_l} + B^{*switch}_l.$$

The forward and backward passes of the FB algorithm require the storage of all α_{s_l} and β_{s_l} values in an $N \times L$ matrix for each observation (for production of the γ_{s_l} results according to eq. (III.9). However, in the case of the SF algorithm (used to compute $\mathrm{P}(h_{1:L})$ or $\mathrm{P}(s_L \mid h_{1:L})$ only), the intermediate $\alpha_{s_{1:l}}$ do not have to be stored, only α_{s_l} (α_{s_L} at the end of the algorithm).

Algorithm 4 The sequential forward pass of the FB algorithm under the LS model

init: Calculate the α'_{s_1}, $A^{stay}_{s_1}$, $A^{*switch}_1$ values for all s_1;
for all inference haplotypes $h_1, ..., h_K$ **do**
  for all loci $l \in \{2, ..., L\}$ **do**
    Initialise $c^{\alpha}_l = 0$;
    for all possible 'present' states $s_l \in \{1, ..., N\}$ corresponding to $H_1, ..., H_N$ **do**
      Calculate $\alpha_{s_l} = \mathrm{P}(h_l \,|\, s_l)\, (A^{stay}_{s_l} + A^{*switch}_l)$ and save;
      Update $c^{\alpha}_l += \alpha_{s_l}$;
    end for
    Save the normalising constant c^{α}_l;
    for all possible 'present' states $s_l \in \{1, ..., N\}$ corresponding to $H_1, ..., H_N$ **do**
      Normalise α_{s_l} to get α'_{s_l} and save;
      Pre-calculate $A^{stay}_{s_{l+1}}$ with α'_{s_l} and save for use in iteration $l+1$;
    end for
  end for
end for

Algorithms 4 and 5 present a version of the forward and backward passes of the FB algorithm that achieve the $\mathcal{O}(N)$ algorithmic speedup under the LS model. It is clear from the pseudocodes that under the LS model the inner loop has to be broken up for calculating the scaling factors in order to preserve the algorithmic speedup and retain numerical stability at the same time. It is also clear that the first and the last inner loops of the backward pass can be merged - the pseudocode reflects this structure for clarity.

Algorithm 5 The sequential backward pass of the FB algorithm under the LS model

1: **init**: Calculate the β'_{s_L}, $B^{stay}_{s_L}$, $B^{*switch}_{L}$ values for all s_L;
2: **for all** inference haplotypes $h_1, ..., h_K$ **do**
3: **for all** loci $l \in \{1, ..., L–1\}$ **do backward**
4: Initialise $c^{\beta}_{l} = 0$;
5: **for all** possible 'present' states $s_l \in \{1, ..., N\}$ corresponding to $H_1, ..., H_N$ **do**
6: Calculate $\beta_{s_l} = B^{stay}_{s_l} + B^{*switch}_{l}$ and save;
7: Update c^{β}_{l} += β_{s_l};
8: **end for**
9: Save the normalising constant c^{β}_{l};
10: Initialise the $B^{*switch}_{l–1}$ estimate as 0;
11: **for all** possible 'present' states $s_l \in \{1, ..., N\}$ corresponding to $H_1, ..., H_N$ **do**
12: Normalise β_{s_l} to get β'_{s_l} and save;
13: Pre-calculate $B^{stay}_{s_{l–1}}$ with β'_{s_l} and save for use in iteration l–1;
14: Calculate $B^{switch}_{s_l}$ with β'_{s_l} and update $B^{*switch}_{l–1}$ += $B^{switch}_{s_l}$;
15: **end for**
16: **end for**
17: **end for**

The parallel SF and FB algorithms under the LS model

Parallelisation of the outer loops (over k) and inner loops (over s_l) in the main recursions of the forward and backward passes are done similarly to the parallelisation of the Viterbi algorithm. As expected, the critical steps are the summation reductions over s_l when calculating $A^{*switch}_{l+1}$, $B^{*switch}_{s_{l–1}}$ and the scaling factors.

Algorithms 6 and 7 present the parallel forward and backward passes under the LS model. In practice, the inner parallel operations (including the parallel reductions) can be performed with a single kernel. It is important to note that the reduction operations in line 7 of Algorithm 6 as well as lines 7 and 13 of Algorithm 7 each represent $\mathcal{O}(\log N)$ operations.

To complete the FB algorithm, one has to calculate the γ_{s_l} values and the products of the normalising factors from the forward and backward passes (according to eq. (III.9)). In practice, it is beneficial to perform these calculations during the backward pass, as soon as all required values are available, to save storage and memory access operations.

Algorithm 6 A parallel forward pass of the FB algorithm under the LS model

init: Calculate the α'_{s_1}, $A^{stay}_{s_1}$, $A^{*switch}_1$ values for all s_1;
parallel for all observed haplotypes $h_1, ..., h_K$ **do**
  for all loci $l \in \{2, ..., L\}$ **do**
    parallel for all 'present' states $s_l \in \{1, ..., N\}$ corresponding to $H_1, ..., H_N$ **do**
      Calculate $\alpha_{s_l} = \mathrm{P}(h_l \,|\, s_l)\,(A^{stay}_{s_l} + A^{*switch}_l)$ and save;
    end parallel
    Calculate the normalising constant $c^{\alpha}_l = \sum \alpha_{s_l}$ with parallel reduction and save;
    parallel for all 'present' states $s_l \in \{1, ..., N\}$ corresp. to $H_1, ..., H_N$ **do**
      Normalise α_{s_l} to get α'_{s_l};
      Pre-calculate $A^{stay}_{s_{l+1}}$ with α'_{s_l} and save for use in iteration $l{+}1$;
    end parallel
  end for
end parallel

Algorithm 7 A parallel backward pass of the FB algorithm under the LS model

init: Calculate the β'_{s_L}, $B^{stay}_{s_L}$, $B^{*switch}_L$ values for all s_L;
parallel for all inference haplotypes $h_1, ..., h_K$ **do**
  for all loci $l \in \{1, ..., L{-}1\}$ **do backward**
    parallel for all 'present' states $s_l \in \{1, ..., N\}$ corresponding to $H_1, ..., H_N$ **do**
      Calculate $\beta_{s_l} = B^{stay}_{s_l} + B^{*switch}_l$ and save;
    end parallel
    Calculate the normalising constant $c^{\beta}_l = \sum \beta_{s_l}$ with parallel reduction and save;
    parallel for all 'present' states $s_l \in \{1, ..., N\}$ corresp. to $H_1, ..., H_N$ **do**
      Normalize β_{s_l} to get β'_{s_l} and save;
      Pre-calculate $B^{stay}_{s_{l-1}}$ with β'_{s_l} and save for use in iteration $l{-}1$;
      Calculate $B^{switch}_{s_l}$ with β'_{s_l};
    end parallel
    Calculate $B^{*switch}_{l-1}$ with par. red. from $B^{switch}_{s_l}$ and save for use in iter. $l{-}1$;
  end for
end parallel

2.2.2 Algorithm analysis and considerations for implementation

Suitability for parallelisation

It is clear from Section 2.2.1 that the SF and FB algorithms are suitable for parallelisation over both observation sequences and states under the LS model, similarly to the Viterbi algorithm. Parallelisation over states requires frequent communication between parallel threads, hence it is most suitable for GPUs than alternative architectures, as was the Viterbi algorithm.

Compared to the Viterbi algorithm, the necessity of scaling and normalisations make coding and optimisations slightly more complicated for the forward and backward passes. Normalisations require additional reduction steps, hence implementations require special care to retain efficiency.

A further difficulty with the parallelisation of the FB algorithm is that it has higher memory requirements, while parallelisation of the SF algorithm is particularly promising due to its near-negligible memory requirements.

Space requirements

Just as in the case of the Viterbi algorithm, the analysis of space requirements is vital when assessing the practical applicability of the SF and FB algorithms.

The SF algorithm has much lower space requirements than the Viterbi or the FB algorithms because the α_{s_l} need to be stored only for the last l at any given point in the algorithm. Consequently, most of the memory is required for the observation sequences $\mathcal{H}'$, the reference sequences $\mathcal{H}$ and the intermediate results $A_l^{*switch}$ and $A_{s_l}^{stay}$, amounting for a total of $\big(K{\times}L+N{\times}L+2(K{\times}N)\big)$ space. The significantly lower memory requirements make the parallelisation of the SF algorithm particularly promising. Low memory requirements mean negligible streaming overhead (see under streaming later in this section) and also imply that access to the the relatively slow GMEM is greatly reduced.

The FB algorithm has significantly higher requirements. The dominating data structures are the result tables (α_{s_l}, β_{s_l} and γ_{s_l} stored for all l, all states ($s \in \mathcal{H}$) and all observations ($\boldsymbol{h_k} \in \mathcal{H}'$)), which amount to $K{\times}N{\times}L$ floating point values. These floating point values can be expected to have 2-4 times the space requirement compared to the short integers usable for the backtrack matrix in the Viterbi algorithm. The scaling factors also need to be stored for each l and each observation, but their required space is negligible in comparison.

For practical data size limitations and host memory space requirements, see Tables III.2 and III.3 in Section 2.2.3.

Block-partitioning for CUDA streaming

A serious restriction when developing for GPUs is that GMEM, CMEM and SMEM are limited (typically in the range of 1-12GBs of GMEM, 64kBs of CMEM and 16-48kBs of SMEM are available in current GPUs). Overcoming this obstacle is absolutely necessary if the FB algorithm is intended to cope with the typical dataset sizes in genetics applications, but in the case of very large data sets the SF may also face space limitations on the GPU.

Algorithm 8 Streaming for the forward pass of the FB algorithm: after the observed sequences $\mathbf{h_k} \in \mathcal{H}'$ are partitioned into sub-sequences, the forward pass is run sequentially on each observation part to overcome the device memory (GMEM) and shared memory (SMEM) space constraints of GPUs.

1: **init**: α_{s_1} , $A_{s_1}^{stay}$ and $A_1^{*switch}$ values for all s_1;
2: Define P parts of length $L_p = L/P$ over the loci $\in \{1, ..., L\}$;
3: **for all** parts $p \in \{1, ..., P\}$ of the inference sequences **do**
4: Move all data required for the current part j to the device;
5: Run the forward pass on part j with given starting A^{stay} and $A^{*switch}$ values;
6: Keep the final $A_l^{*switch}$ and $A_{s_l}^{stay}$ results on the device;
7: Send the α_{s_l} matrix corresponding to the current section p of L_p loci to the host;
8: **end for**
9: **merge**: Simply concatenate the results along l to get α_{s_l} for $[1, L]$.

The increased space requirements of the FB algorithm require the application of CUDA streaming, a platform-provided facility that allows the parallel execution of kernels and two-way asynchron data transfers between host and device (D2H and H2D transfers). Streaming supports the division of large problems into parts, which are then executed sequentially.

Observing the fact that the SF and FB algorithms (as well as the Viterbi algorithm) have to proceed sequentially over all loci irrespective of any parallelisation, it is clear that computation can be divided into sequential sub-tasks that may have sufficiently low memory requirements to fit onto a GPU. The algorithms can be separated into a number P of small runs on $L_p = L/P$ long parts of the observations. With a sufficiently large number of sub-tasks, the memory requirements can be reduced almost arbitrarily ($L_p \geq 2$ compared to potentially $L \sim 10^6$). This partitioning method is a perfect fit for CUDA streaming.

The incorporation of streaming requires a few modifications in the SF and FB algorithms. The streaming procedure will be demonstrated for the forward pass of the FB algorithm but is similarly applicable to the backward pass and the SF and Viterbi algorithms. The first streaming run of the forward pass on the first part of the observation sequence needs to return its α_{s_l} result matrix to the host and save its final $A_l^{*switch}$ and $A_{s_l}^{stay}$ values on the device in GMEM. The streaming run on the second part will use these $A_l^{*switch}$ and $A_{s_l}^{stay}$ values, return its own α_{s_l} matrix and find its own final $A_l^{*switch}$ and $A_{s_l}^{stay}$ values, which can then be used for the next part. This process produces a set of α_{s_l} matrix parts that can be concatenated into the complete α_{s_l} matrix equivalent to what the results of a normal forward pass would have been (see an outline of the procedure under Algorithm 8). Streaming is very similar to the normal checkpointing methods used for HMM algorithms (see Grice et al. (1997) and Tarnas and Hughey (1998) for checkpointing for sequential HMM algorithms).

Analysis

It is clear that with the use of streaming it is possible to eliminate the hard GMEM and SMEM memory constraints completely and parallel implementations of HMM algorithms can be made applicable for datasets with large L from the GPU's perspective.

It is also clear that streaming does not result in any acceleration for the calculations but rather introduces some extra instructions due to the necessary data assembly steps. The cost of partitioning and merging is minor or negligible in most cases.

The power of CUDA streaming comes from the possibility of concurrently executing GPU and CPU calculations with data transfers in both directions and hence 'hiding' most of the time required for data transfers (NVIDIA, 2014). In the case of P parts, P or $P{+}1$ or $P{+}2$ streaming iterations are performed, with data transfers potentially preceding the first and succeeding the last kernel run. Since the forward and backward passes of the FB algorithm are either input or output intensive and the SF algorithm requires minimal data transfer, the complete streaming run-times are roughly proportional to

$$T^{FB}_{streaming} \propto (P{+}1)\, L_p = L + L_p \tag{III.14a}$$

$$T^{SF}_{streaming} \propto (P)\, L_p = L \tag{III.14b}$$

where L corresponds to the amount of calculation necessary for the algorithm and L_p to the amount of data transfer not hidable by streaming. In theory, the consequence is that setups with large P and low L_p should be favoured for the FB algorithm, but in practice performance decreases if L_p is overly low, because the overhead of kernel launches becomes more significant (see the empirical performance results for the streaming SF and FB algorithms in Section 3.2.2).

Besides L_p, the efficiency of streaming is also determined by the balance between the overlapped kernel run and data transfer times. In cases where the data that have to be transferred are large (when N and K are large), the transfer times can dominate execution time and, hence, can reduce or completely negate the efficiency of kernels.

CPUs and GPUs are expected to become more closely coupled in the future. Data transfer costs between host and device and the strict global memory restrictions can be expected to be much lower or even irrelevant, with possibly no need for streaming.

Using virtual threads

Another CUDA restriction is that only 1024 threads are allowed in each thread block. In order to overcome this limitation, it is possible to use a large number of 'virtual threads'

over a smaller number of actual CUDA threads. Virtual threads can be divided into bundles of a set number (TMAX) of threads. The virtual bundles share the actual CUDA threads. Calculations can be performed only on one bundle at a time, but bundles can be cycled through, swapping or 'paging' them in and out sequentially - similarly to how virtual memory works in modern operating systems. Paging works by saving and loading key thread variables.

Paging introduces some forced sequentiality to implementations because computation on different bundles has to be performed sequentially and also introduces extra memory operations (usually over GMEM). However, virtual threads also allow for complete flexibility on the choice of TMAX, which may be useful for optimisation.

Theoretical analysis of computational complexities and run times

The approximate calculation of execution times and computational complexities follows the principles used for the Viterbi algorithm (Section 2.1.3) and the results are identical.

The time complexity of the general sequential FB algorithm is $\mathcal{O}(KLN^2)$ and that under the LS model is $\mathcal{O}(KLN)$. It is clear that the degree of parallelism in the parallel algorithms presented here for the LS model is $K{\times}N$-fold for all forward and backward passes. Because of the necessity of parallel reductions in the 'inner loop' (see Algorithms 6 and 7), the parallel algorithm requires $\mathcal{O}(\log N)$ times extra computation, leading to a total of $\mathcal{O}(KLN \log N)$ computational complexity. Therefore the achievable theoretical acceleration of the SF and FB algorithms are $\mathcal{O}(KN/\log N)$ with a theoretical runtime of $\mathcal{O}(L\ \log N)$.

When compared to the parallel forward passes, parallel backward passes require more operations in each iteration because reductions have to be performed twice, one of which isn't required in the forward pass (see equation (III.12)). As a result, there may be a significant constant factor by which the backward pass can be slower (which is proven by practical results - see Section 3).

2.2.3 Implementations, optimisations and limitations

The sequential SF and FB algorithms were implemented in MATLAB and C++ according to Algorithms 4 and 5. The parallel Forward and FB algorithms were implemented in CUDA based on Algorithms 6 and 7. The sequential implementations were used for prototyping and preliminary analyses and also served as accuracy and runtime comparison benchmarks. While the sequential C++ implementations were primarily kept as similar to the CUDA versions as possible, they were also moderately optimised to serve as benchmarks in practical acceleration experiments. The CUDA implementations were developed from the sequential C++ codes with the step-by-step introduction of parallelisms over $k \in \mathcal{H}'$ and then $s_l \in \mathcal{H}$.

The parallel FB algorithm was implemented using separate kernels for the forward and backward passes. The parallel SF algorithm and the backward pass of CUDA-Chromopainter (CCP) were implemented using specialised kernels.

For simplicity, observations were 'assigned' to thread blocks and reference sequences were 'assigned' to threads. In order to overcome the maximum thread limit (which would have meant a hard $N \leq 1{,}024$ restriction), all parallel implementations used virtual threads. Communications (tree-structured parallel reduction operations) between virtual threads were performed over GMEM in order to keep the algorithm stable, which resulted in some moderate, unavoidable performance loss. In other CUDA HMM applications (e.g. Cartey et al. (2012)), the GPU's warp size (usually 16 or 32 in current GPUs) was chosen for page size (the maximum number of actual CUDA threads used, TMAX). Interestingly, for the achieved implementations for HMM algorithms under the LS model the warp size did not prove to be optimal. Empirical results showed (data not presented here) that the use of TMAX=256 actual CUDA threads showed the most reliable all-round performance across various different data sizes. This is likely because larger numbers of threads are less efficient to manage with the CUDA scheduler and smaller numbers of threads require certain reduction buffers to be large (in SMEM), which means that the L1 cache has to be limited to 16kBs so that SMEM could be set to 48kBs. Accordingly, the page size was simply fixed at TMAX=256.

In order to achieve practical applicability, all parallel implementations incorporated streaming. As argued in Section 2.2.2, theoretically the optimal choice would be to use small L_p streaming block sizes for the FB algorithm in order to reduce the amount of data transfer that can not be hidden by streaming (see equation (III.14a)). Empirical experiments proved that performance is indeed lower when L_p is not significantly smaller than L. The experiments also revealed that performance drops when $L_p \times N$ is small, in which case the host-level synchronisations, the overheads and data transfers related to kernel launches become significant (see Figures III.6, III.7 and III.10 in Section 3.2.2). When L_p and/or N are sufficiently large, there are enough calculations in kernels to hide the effect of these overheads.

For each kernel implementation, the optimal size of streaming blocks L_p are estimated in run-time as the largest L_p for which all data fit into SMEM and GMEM, given K and N. After extensive testing and assuming $L \gg 500$, the streaming block size L_p was capped at 500 because higher values were not expected to yield performance improvements (see Figure III.10). Explicitly limiting L_p allowed for the use of CMEM in further optimisations (CMEM stored the transition probabilities of size $16L_p$).

Optimisations

With the application of streaming, optimisations had to be performed on two interlinked levels - on kernel level and streaming level. On the kernel level the balance between memory operations and computations are the key for performance. Accordingly, memory operations were minimised and SMEM was used wherever possible to reduce GMEM use.

On the streaming level, best performance is achieved whenkernel execution times (which depend on the kernel-level optimizations) and data transfer times (H2D and D2H) are in balance. For HMM algorithms, this balance largely depends on the dimensions of the datasets used and the exact parameterizations of the streaming procedure. In order to reduce data transfer, the results of the forward and backward passes were stored as single-precision floating point numbers and intermediate values were kept in device memory whenever possible. Naturally, streaming-level (host-level) synchronisations were also used as little as possible.

All kernels and host functions were developed and empirically assessed under various different optimisation strategies until relatively high efficiency was achieved. Performance testing consisted of the measurement of execution times of individual kernel sections as well as streaming-level analysis using the CUDA Visual Profiler.

Minimizsation of the number of required registers did not seem to translate into higher efficiency, while the minimisation of synchronization points and GMEM operations did. Reduction algorithms were performed in multiple tiers. First, across warps of threads (32 threads, 5 iterations), next across warps in virtual thread bundles (TMAX=256 threads, 3 more iterations), third across warp-sized groups of the 256-sized bundles of virtual threads (8192 threads, 5 iterations) and at last across warps again. All reductions were performed in SMEM, with only a single pair of GMEM operations required between tiers 2 and 3. This tiered reduction structure allowed cross-warp reductions on three levels, greatly reducing the number of required thread synchronisations in the reduction operations.

In the final implementations, kernel launches were minimised, all memory layouts were coalesced, most communication between threads were facilitated through SMEM with minimal synchronisations, branching was minimised and the calculation of γ_{s_l} results were built into the kernels of the backward pass.

The final versions of the SF and FB algorithms work with K blocks (corresponding to the number of inference or recipient haplotypes) and 256 threads in each block. In both algorithms, the switching and staying probabilities are stored in CMEM, along with all scalar parameters and mutation probabilities. The result tables ($\log(\alpha_{s_l})$, $\log(\beta_{s_l})$ and $\log(\gamma_{s_l})$) in the FB algorithm are stored as in GMEM as single-precision floating point numbers, while scaling factors and intermediate results are stored in SMEM with double-precision.

Table III.2: Hard dataset size limits

Method	K	N	L	L_p
SF algorithm	65,536*	65,536**	no hard limit	2-500***
FB algorithm - fwd pass	65,536*	65,536**	no hard limit	2-500***
FB algorithm - bwd pass	65,536*	65,536**	no hard limit	2-500***
FB algorithm - bwd pass with Chromopainter's chunkcount calculations	**7,186**	65,536**	no hard limit	2-500***

* The 65,536 is the absolute limit for the number of blocks on current CUDA GPUs, which is also expected to increase in the future.

** 65,536 is the highest number a short integer is guaranteed to be able to hold on all platforms.

*** The minimum length L_p of the streaming blocks is 2 and the maximum of L_p was fixed at 500 for optimisation purposes.

$K = |\mathcal{H}'|$ is the number of inference sequences, L is the number of loci (the length of inference and reference sequences) and $N = |\mathcal{H}|$ is the number of reference sequences

This setup aims to optimisze kernel performance and minimise data transfers while still remaining numerically stable, irrespective of data sizes. Although working with double-precision floating point numbers greatly reduces the performance of kernels (about 2-fold), it is necessary to remove the possibility of arithmetic underflows.

Both C++ and CUDA implementations were compiled using optimisation level O3 and the parallel implementation was compiled under CUDA version 5.5.22 and using 'fast math'.

Limitations

The practical data size limitations and host memory space requirements are shown in Tables III.2 and III.3. Hard limits are absolute limits on each dataset dimension (K, N, and L) individually. Most are imposed by the CUDA framework itself (e.g. a maximum of 65,536 blocks usable in the the first CUDA block dimension), others are imposed by the memory requirements of the optimization schemes applied. Soft limits are encountered when the overall GMEM, SMEM, CMEM or host memory usage - determined by the combination of N, K and L jointly - exceeds the available memory space (CMEM, SMEM, GMEM or host memory). Soft limits restrict the combinations of N, K and L which the algorithms can handle. Table III.3 presents the maximum sizes of various datasets for current high-end GPU systems. For the final implementations, it is the host memory that limits the data size, not the GPU. The figures also show that the FB and SF algorithms are readily applicable to realistic genetic data sets.

Table III.3: Soft dataset size limits in practical use for individual runs

Method	Dataset	K	N	L	L_p	SMEM (B)	GMEM (MB)	CMEM (B)	Host mem. (MB)
FB alg.	symmetric	20,000	20,000	55	2	5,122	**12,208**	78	**93,083**
FB alg.	max N	32	65,535	11k	500	6,644	**12,110**	8,046	**89,423**
FB alg.	max L	32	128	6mill	500	4,604	24	8,046	**95,924**
FB alg.	realistic	32	4000	190k	500	4,724	739	8,046	**94,248**
SF alg.	symmetric	40,000	40,000	350k	351	4,447	**12,288**	8,046	**90,047**
SF alg.	max N	24,000	65,535	350k	500	4,596	**12,128**	8,046	**95,791**
SF alg.	max L	256	2,200	20mill	500	2,676	8	8,046	**94,846**
SF alg.	realistic	5,000	5,000	5mill	500	2,804	205	8,046	**96,226**
CCP	max K	7,186	32,000	100	2	5,122	**12,281**	**65,534**	**94,745**
CCP	max N	32	65,535	11k	500	6,644	**12,158**	8,302	**89,439**
CCP	max L	32	128	6mill	500	4,604	24	8,302	**95,925**
CCP	realistic	32	4000	190k	500	4,724	742	8,302	**94,249**

$K = |\mathcal{H}'|$ is the number of inference sequences, L is the number of loci (the length of inference and reference sequences) and $N = |\mathcal{H}|$ is the number of reference sequences

The above dataset sizes assume a relatively high performance system by current standards, with Host Memory = 96GB and with GPUs of GMEM=12GB. CMEM and the maximum of SMEM are assumed to be the usual 16kB and 48kB, respectively. The figures aim to demonstrate limits and not practical configurations.

The figures in bold highlight where memory limits are reached. Generally, K and N are restricted by SMEM and L is restricted by host memory only.

The 'realistic' sets for the FB and CCP methods suppose that the user can divide the observation sets into blocks of 32.

After the application of streaming and virtual threads, the following limitations remained for the final versions of the CUDA SF and FB algorithms:

- The current implementations assume and enforce $K<65,536$ and $N<65,536$, mainly for simplicity. While it is possible to use more CUDA blocks and more virtual threads, and the block sizes allowed by CUDA may also increase in the future, implementations require more work to ensure that no values overflow or underflow at large N and K in order to ensure that the implementations are stable on all systems. For currently existing data sizes these limitations should not be a problem, but in the close future (with sample sizes between $10^5 - 10^6$) they may become relevant.
- The SF and FB algorithms have further limitations on applicability due to soft memory constraints - when the GPU's SMEM and GMEM and the host's operative memory are not large enough for combinations of K, N and L. The GPU-related limitations will likely be reduced and may completely disappear with the increase of GPU memory

sizes. However, the most constricting limitation is the availability of host memory. Increases in host memory can also be expected in the future but large-memory systems already exist (e.g. systems with >1000GBs of memory). In the absence of the above resources, datasets may be divided across $k \in \mathcal{H}'$ without loss of information, however, K should never be too low (e.g. <32) and should be a multiple of 32 if possible in order to retain high performance (see Section 3.2.2 for empirical results).

- The page size is fixed at 256 threads. While runs with 256 threads showed good and robust performance, 256 is sub-optimal for certain datasets. Experimental results show that using 256 as the page size reduces performance by 7% at most when compared to the optimum (data not presented).

- When N is large, the data transfers can be slow and can limit the performance of streaming for the FB algorithm. Although not convenient, this limitation can be countered in most cases by dividing analyses across $\mathcal{H}'$. Naturally, the parallel SF algorithm does not have limitations related to data transfer times.

- It is quite likely that some optimisation opportunities have been missed and that performance can be improved with further analysis and work, although the results are quite significant with the current implementations already (see Section3.2.2).

2.3 Applications

For the purpose of demonstrating the practical relevance of the parallel SF and FB algorithms, two example applications have been implemented.

The 'ChromoPainter' method (Hellenthal et al., 2014; Lawson et al., 2012) was the first choice for parallelisation because of its success, because it is very close to the LS model and also due to inspiration from the variant calling algorithm presented in Chapter II. ChromoPainter is primarily used to determine recent common ancestry between haplotypes or groups of haplotypes. It was developed to be used in connection with the method fineSTRUCTURE (Lawson et al., 2012) to determine population structure and history. The original method takes days or weeks to run on large genetic datasets and is, hence, a good candidate for demonstrating the value of parallelisation.

The second chosen application was large-scale parallel likelihood calculation under the LS model. It is used to map the likelihood in a given parameter space for 1) estimating the joint maximum likelihood and 2) screening datasets based on the distribution of MLEs of various model parameters across haplotypes. The aim with this application is to demonstrate the applicability of the CUDA SF algorithm for various purposes.

2.3.1 Chromosome painting

The original Chromopainter (OCP)

Chromosome painting is used for uncovering and characterising shared ancestry between genetic sequences. In essence, each haplotype is modelled with sections of other haplotypes by direct application of the LS model (Lawson et al., 2012). The primary purpose of the method is to calculate the expected 'chunk counts', the number of shared regions between haplotypes (see Figure III.2).

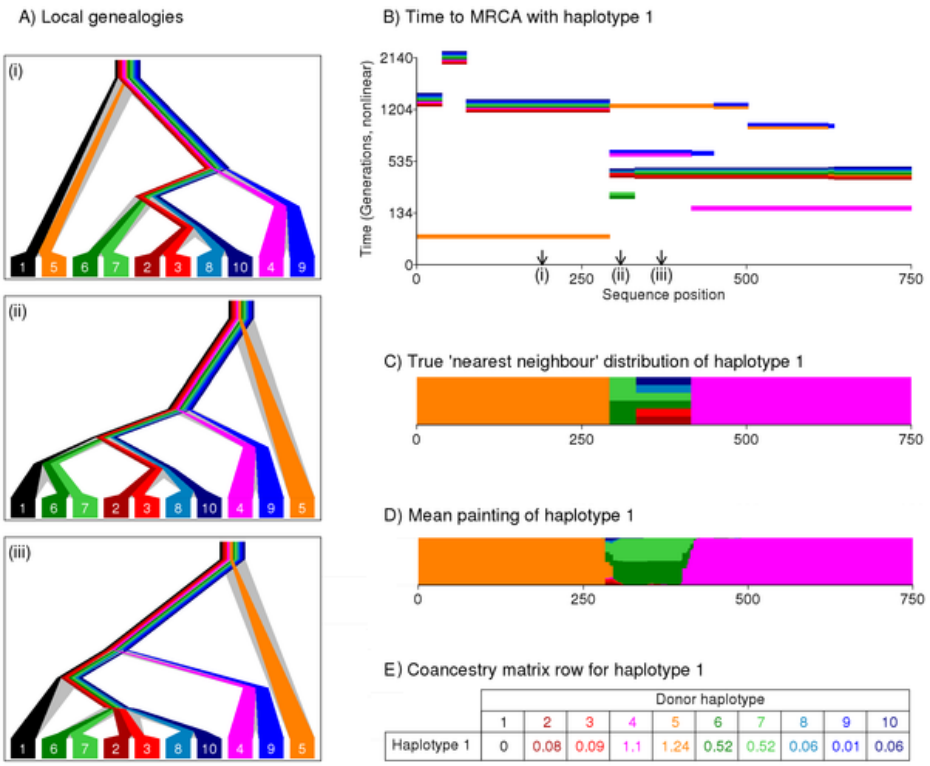

Figure III.2: Demonstration of chromosome painting - determining coancestry between haplotypes (modified from Lawson et al. (2012)). Haplotype 1 (black) is 'painted' with other haplotypes. The true genealogies are shown in (A). The times to the most recent common ancestor (TMRCA) between haplotype 1 and the others as a function of sequence position are shown in (B). The true distribution of the 'nearest neighbour' haplotype is shown in (C). The results of the painting process are the estimated nearest neighbour distribution (D) and the row of the result coancestry matrix or copied 'chunk counts' corresponding to haplotype 1 (E).

The close agreement of the original Chromopainter (OCP) with the the original LS model allowed its parallel implementation to be based on the parallel FB algorithm without major restructuring. Chromopainter employs a fine-scale recombination map, can work with a single global mutation rate and uniform initial distributions. It can also work with a single global recombination rate scaler, which is defined slightly differently from that in the original

LS model ($N_e^{OCP}=4N_e^{LS}/(N+K)$. Chromopainter uses Watterson's estimate as the default global mutation rate as in the LS model and uses $N_e^{OCP}=400{,}000/(N+K)$ as the default recombination scaler.

The main output produced by Chromopainter is a matrix of chunk counts with entries $\hat{x}_{kn}$ for each $k \in \mathcal{H}'$ 'recipient' haplotype (the observations) and each $n \in \mathcal{H}$ 'donor' haplotype (the reference haplotypes). These chunk counts are calculated for each recipient k, using the results $\alpha_{s_l}=\mathrm{P}(h_{1:l}, s_l)$ and $\beta_{s_l}=\mathrm{P}(h_{l+1:L}, s_l)$ from the forward and backward passes of the FB algorithm as follows:

$$\hat{x}_{kn} = \frac{\alpha_{s_1}\beta_{s_1}}{\mathrm{P}(D)} + \sum_{l=1}^{L-1} \frac{1}{\mathrm{P}(D)} \left[\alpha_{s_{l+1}}\beta_{s_{l+1}} - \alpha_{s_l}\,\beta_{s_{l+1}}\,\mathrm{P}(h_{l+1}\,|\,s_{l+1}=n)\,\exp(-\rho_l)\right] \quad \text{(III.15)}$$

where $\hat{x}_{kn}$ is the posterior expected number of chunks copied from donor haplotype $\boldsymbol{h_n} = \{h_{n1}, .., h_{nL}\} \in \mathcal{H}$, the constant $\mathrm{P}(D)$ is calculated from normalising α_{s_L}, the term $\mathrm{P}(h_{l+1}\,|\,s_{l+1}=n)$ is the emission probability and $\exp(-\rho_l)$ is the probability that no state transition occurs at l.

CUDA-Chromopainter (CCP)

The primary functions of Chromopainter have been reproduced in MATLAB, C++ (in sequential versions) and CUDA (in a parallel version) for prototyping, preliminary analyses and benchmarking purposes. The sequential and parallel implementations of the FB algorithm provided the starting points for development.

The first versions of the sequential Chromopainter carried out the forward and backward passes and the chunk count calculations separately. Later versions integrated chunk count calculations into the backward pass, similarly to how it is performed in the original Chromopainter (OCP), where chunk count calculations are integrated into the backward recursions.

The calculation of the expected chunk counts did not require changes in the structure of the parallel design of the FB algorithm. However, small important modifications were required to retain performance.

Results from the forward pass had to be made available on the GPU and calculation order and synchroniszations were slightly rearranged to better overlap memory operations with calculations. The $\mathrm{P}(D)$ had to be stored in CMEM for all observations, which introduced a limit $K \leq 7,186$ (see Table III.2) but also minimised the performance loss.

The final version of CUDA-Chromopainter (CCP) was moderately optimised and implemented with two modes, one where the inference set and the reference set are the same (the '`-a 0 0`' option in the OCP) and one where they are given separately. The original Chromopainter method was also re-written to include only the calculations necessary to compute chunk counts, thus providing an additional, 'fair' benchmark (referred to as 'Minimised OCP') for acceleration evaluations.

The final limitations of CCP:

- The current CCP has similar soft and hard limits of applicability as the parallel FB algorithm, with the additional limit of $K \leq 7,186$ (see Table III.2). The added limit is not expected to be relevant in most cases for the currently available memory and data sizes. It can also be bypassed by running the method on subsets of $\mathcal{H}'$, which is expected anyway in order to fit the data into host memory and to achieve maximum performance. Alternatively, the implementation can be changed in order to allow large numbers of K at the cost of some performance decrease.
- The current CCP does not include functions for calculating 'chunk lengths', does not perform sampling and does not allow for EM estimations as the complete OCP method does. It also does not handle all the user options available in the OCP.

2.3.2 Large-scale likelihood calculation

The observation that the parallel SF algorithm is significantly faster than the parallel FB algorithm in terms of execution time (see Section 3.2.3) inspired the exploration of applications for large-scale likelihood calculations. Since the SF algorithm can calculate the log-likelihood for thousands of haplotypes using data from thousands of whole chromosomes ($L \gg 100k$) in a matter of seconds or minutes, it becomes feasible to experiment with applications where such calculations are repeated in great numbers.

Mapping the log-likelihood in the parameter space of the LS model

The original Chromopainter uses the EM algorithm to estimate model parameters (mutation rates, copying proportions and the effective population size) using a small number (default 10) of complete OCP 'runs' for searching for the MLE (locally). While the EM-based MLE search algorithm in the OCP generally gives stable and reliable results, it may be regarded as sub-optimal for the purpose of parameter estimation for two reasons. First, theoretically the EM algorithm can be stuck in local maxima. Second, the OCP is not designed for efficiently performing repetitive calculations and is often impractical for the task due to its lengthy execution.

The simplest method to map the likelihood is to simply re-run the SF algorithm a large number of times, calculating the log likelihood $\log\big[\mathrm{P}(\boldsymbol{h_k}\,|\,\lambda)\big]$ at various parameter configurations. By evaluating the log-likelihood at given 'grid' points in the parameter space, one can plot the surface of the log-likelihood for visual assessment. Such assessments can give indications on whether the likelihood is multi-modal (and hence validate or invalidate the use of EM-based algorithms) and can provide a feel for how the LS model copes with (modelling) the dataset at hand (in the presence of sequencing errors, very similar haplotypes, etc.).

The current example implementation explores a user-given section of the 2D parameter space $M_g \times N_e$ at given grid points where M_g is the global mutation rate and N_e is the global recombination rate scaling factor.

Estimating the maximum likelihood

It may be a crude and inefficient brute-force method, but the above grid-based results can be directly used to estimate the likelihood in two simple steps. First, the log-likelihood mesh can be refined by some interpolation algorithm (e.g. cubic interpolation) and then the maximum of the new mesh can be taken as the MLE.

An advantage of this approach is that the joint and the individual MLEs can all be estimated this way with no further effort and without much further computation.

Screening genetic datasets based on the distribution of LS parameter MLEs

The parallel SF algorithm calculates the log-likelihood for each haplotype separately and calculates the population-level joint log-likelihood from the individual results. It is possible, however, to explore the individual log-likelihoods separately as well.

With multiple evaluations of the individual log likelihoods, it is possible to create a grid for the individual log-likelihoods in the $M_g \times N_e$ space. From the grid, the individual MLEs and the corresponding log-likelihoods can be estimated after interpolation.

Given a set of MLEs for the global mutation and recombination rate scaler for each haplotype, it is possible to make some basic judgements about individual haplotypes in a dataset. For instance, in order to improve reference datasets, it is possible to filter out outlier haplotypes with overly low likelihood and overly high M_g.

Limitations

While the grid method is applicable for mapping the likelihood for visual assessment, theoretically the EM algorithm is more efficient for finding the MLE than a grid-based approach. Since the EM algorithm can also be expected to find the MLE given the smoothness demonstrated by the grid-based assessment (see results in Section 3.3.2), the implementation and application of different parallel EM algorithms may be more adequate for finding the individual and joint MLEs.

3 Results

This section presents validation, performance and application results for the CUDA implementations of parallel HMM algorithms under the LS model.

3.1 Validation: the accuracy and numerical stability of parallel implementations

Achieving accuracy, consistency and stability are not as straightforward with parallel code as they are in the case of sequential programming. Different parts of a parallel algorithm can execute in any order within the boundaries set by explicitly programmed synchronisation constraints. If code permits, two consecutive parallel runs may have completely different results. Since monitoring execution progress can interfere with execution order, validation and debugging tests can often also be unreliable. Furthermore, arithmetic stability can be lower in GPUs compared to CPUs, especially when using single-precision floating point numbers or 'fast math' for high performance. For these reasons, extensive validation tests were performed to prove the accuracy, stability and consistency of results.

The accuracy evaluations of CUDA implementations were conducted in multiple stages. Early tests during development were performed on manually crafted test cases of different $\mathcal{H}'$ and $\mathcal{H}$ sets and recombination maps. The results of the CUDA Viterbi, SF and FB algorithms were compared to those of the corresponding sequential C++ and MATLAB implementations. Later tests used semi-synthetic and real data sets.

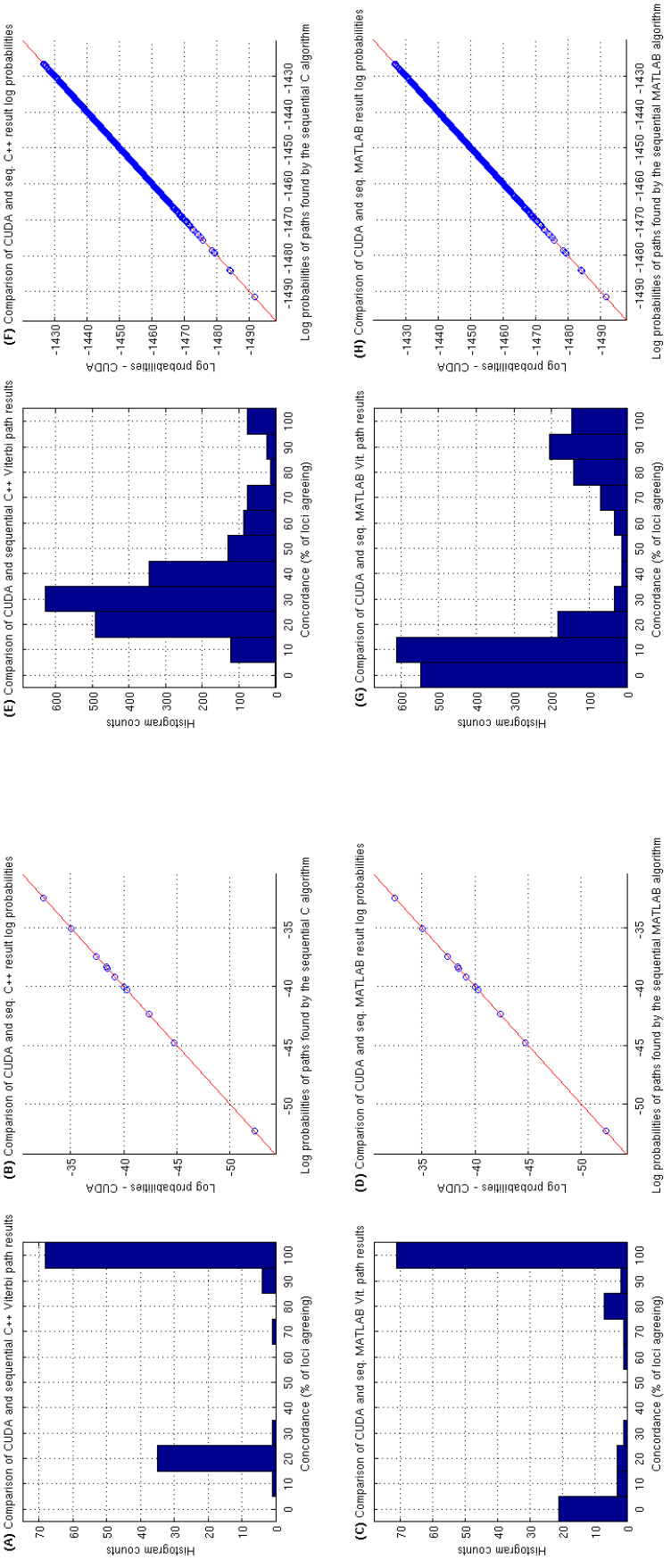

Figure III.3: Comparison of results between different implementations of the Viterbi algorithm under the LS model. The histograms **(A, C)** and the corresponding scatter plots **(B, D)** (left present) comparison results from experimental runs with a batch of 10 small synthetic datasets, all with $K = 11$, $L = 100$ and $N = 20$. Sub-figure **(A)** presents the distribution of concordance (%) between Viterbi paths found by CUDA and sequential C++ implementations and **(B)** the agreement between calculated Viterbi path log-probabilities. **(C)** and **(D)** show the distribution of concordance percentages between the CUDA and MATLAB results and the corresponding scatter plot of Viterbi path probabilities. The subfigures on the right **(E-H)** correspond to comparisons on a batch of 20 slightly larger synthetic datasets with $K = 100$, $L = 1000$ and $N = 20$. **(E-F)** compare results of the CUDA and sequential C++ implementations and **(C-D)** compare results of the CUDA and MATLAB implementations.

3.1.1 Validation results for the parallel Viterbi algorithm

Preliminary observations

Viterbi paths found by the parallel Viterbi algorithm matched for most manually produced datasets, with the exception of some occasional minor differences (these results are not presented). The differences occurred only when the most likely state sequences were not unique. Importantly though, the calculated final probability of the found paths always agreed with good precision. The probability and log-probability results showed occasional, negligible differences. These differences may be due to different floating-point number representations used by the compared implementations (and corresponding different precision errors) as well as the limited precision of some CUDA operations (such as logarithms).

The second stage of accuracy evaluations was performed on semi-synthetic test datasets. The starting point for producing each test set was a reference dataset of $N<120$ haplotypes randomly chosen from the 120 CEU haplotypes of the HapMap 3 data (The International HapMap Consortium, 2007). A given K number of inference haplotypes were generated stochastically according to the LS model, with default parameterisation. In the case of the LS model, it is especially likely that there exist a multitude of equally likely state sequences that may be partially agreeing because of the generally low recombination and mutation rates and potentially high similarity between reference sequences. The results presented in Figure III.3 show that the final probabilities for the found paths are all agreeing, while the paths themselves may be quite different. When multiple paths have equal probability, the choice of the maximum is arbitrary and is often dependent on the platform and memory layouts (specifically the way elements are ordered in arrays).

3.1.2 Validation results for the parallel parallel SF and FB algorithms and CCP

Similarly to the assessment of the Viterbi algorithm, accuracy was quantified as concordance between sequential and parallel implementations of the standalone forward (SF) algorithm, the forward-backward (FB) algorithm and CUDA-Chromopainter (CCP).

Early experiments with the SF and FB algorithms revealed that single precision floating point numbers are sufficient for storing results, but the stability of these algorithms required double-precision floating point numbers for intermediate calculations.

During tests with the final implementations the α, β, γ and chunk count results were all found to be agreeing between sequential and parallel implementations. All results appeared numerically stable and consistent to high precision across multiple runs in all tests.

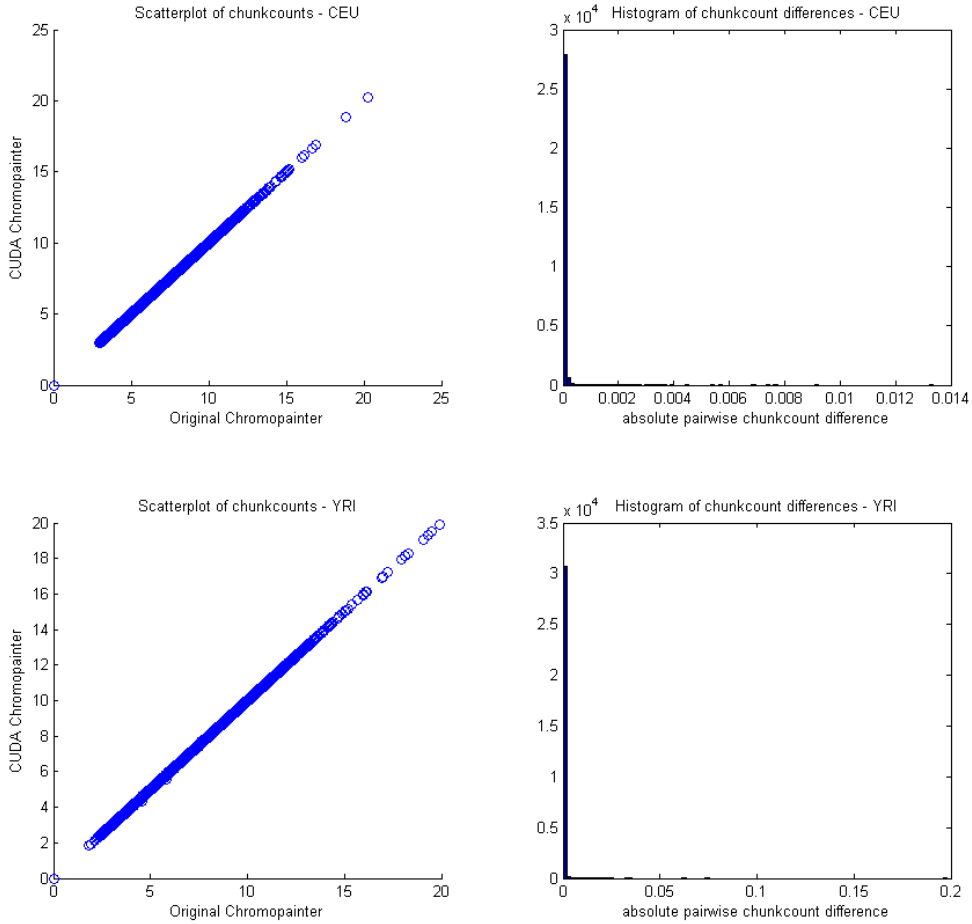

Figure III.4: Accuracy results for the CCP method on phased haplotype data on chromosome 22 from the 1000 Genomes Project. The top row shows agreement results for the the CEU dataset with 170 haplotypes of European ancestry and the bottom row presents results for 176 YRI haplotypes of African ancestry. The scatter plots simply plot corresponding OCP and CCP chunkcount results, the histograms (right) show the distribution of pairwise absolute chunk count differences between OCP and CCP.

The chunk count results of CCP were also compared against results produced with the OCP. These final chunk count agreement results are presented (see Figure III.4) to demonstrate the degree of calculation accuracy of the parallel implementations.

3.2 Performance results

The overwhelming focus of performance analyses was on the dynamic programming recursions, which encapsulate the heart and the most time consuming part of the algorithms and was the only part parallelised in CUDA.

The initialisation, recursion setup and post-processing times are excluded from most analyses

because of 1) their high noise level and dependency on external factors (such as operating system processes), 2) because the significance of these code segments diminish with the use of streaming algorithms and 3) for simplicity and clarity.

Measurements and tools

The execution times of kernels and kernel sections were measured on the host side, using standard C functions (e.g. `clock()`) and CUDA events (e.g. `cudaEventRecord()` and `cudaEventElapsedTime()`). The time measurements were in milliseconds and the accuracy of `clock()` and CUDA time measurements are expected to be 10-30ms and 1ms, respectively. The precision limits are not expected to have significant effect on the results (most measurements are >1000ms) apart from when sequential algorithms are run on very small data sets, in which case 0ms was measured occasionally, or measurement errors were not negligent (<1000ms). Most measurements were performed 5-20 times to minimise the effect of measurement errors and temporary external effects such as server delays as much as possible. Most performance results come from measurement averages from multiple experiments.

Besides direct time measurements, the CUDA Visual Profiler (NVVP) was used for visual assessment, particularly when monitoring the efficiency of streaming and the balance between data transfer times and kernel execution times. In general, data transfer times have been found to be noisy and fluctuate according to environmental variables but their efficiency was not affected by dataset size. On the other hand, kernel execution times were found to be relatively consistent, with their performance having greatly varied with dataset dimensions.

Hardware and datasets used

Every experiment was performed on the Emerald system, the e-Infrastructure South GPU supercomputer (`http://people.maths.ox.ac.uk/gilesm/emerald.html`). The CPUs of the Emerald system are 6-core X5650 Xeon processors with 2.66-3.06GHz clock frequency. The GPUs are 512-core M2090 Fermi models with a 6GB GMEM each.

For convenience, most performance experiments were done on fully synthetic random-generated data sets as the content of datasets is not expected to have a significant effect on execution times. The consistency of parallel runtime results observed throughout all experiments prove the validity of this assumption.

Real data experiments used various haplotype sets from the 1000 Genomes Project. These phased data sets have been produced with the method SHAPEIT (Delaneau et al., 2012) and were pre-processed to exclude monomorphic sites.

Performance metrics

The performance of the parallel HMM methods is strongly effected by the dimensions of the datasets that the methods are run on. In order to conduct a consistent performance analysis and compare results on different data sets, the efficiency metric

$$E = \frac{KLN}{T_{execution}} \tag{III.16}$$

was used (K, L and N are the usual dataset dimensions). This efficiency metric can be interpreted as the speed at which an HMM algorithm traverses through the dynamic programming trellis (or the result tables α_{k,s_l}, β_{k,s_l} or γ_{k,s_l} produced in the FB algorithm). The metric directly corresponds to the computational complexity $\mathcal{O}(KLN)$ of sequential HMM algorithms under the LS model and is the inverse of the constant execution time factor c in $T_{execution} = c\,KLN$.

In this section, performance will be measured by execution time, efficiency (as defined above) and acceleration (simply calculated as $X = T_{sequential}/T_{parallel}$).

3.2.1 Exploratory performance and acceleration results for the parallel Viterbi algorithm

For the exploratory experiments with the Viterbi algorithm the datasets were generated with sizes according to all combinations of $K \in \{1, 64, 128, 256, 512, 768, 1000\}$, $L \in \{100, 200, 300, 400, 500, 600, 700, 800, 900, 1000\}$ and $N \in \{2, 64, 128, 256, 512, 768, 1024\}$. The sequence lengths were limited because streaming has not been implemented for the Viterbi algorithm.

The collected measurement results are presented by the number of states, inference sequences and loci (Figure III.5). The computation times of the CUDA and sequential C++ implementations were compared to characterise the extent of acceleration achievable by the parallel CUDA implementation.

The results show that execution times of the sequential algorithm increase linearly with K, L and N (Figure III.5), as expected. The CUDA recursion times are clearly much lower and also increase linearly with K and L but not exactly linearly with N. The non-linear runtime increase may be attributed partly to the extra $\mathcal{O}(logN)$ factor in computation and also partly to the 'appropriateness' of the number of threads (i.e. how easy or hard it is for the CUDA scheduler to utilise the maximum of GPU resources for a given thread configuration). For

instance, an inappropriate number of threads such as 768 may be too large or not divisible by the maximum number of concurrent threads (which is 1536 in Fermi GPUs when the number of registers used is low). An inappropriate number of threads can lead to having idle threads during execution and/or to serious restrictions for the GPU scheduler. Usually the ideal numbers for block sizes (number of threads) are expected to be somewhere among $\{32, 64, 128, 256\}$ (NVIDIA, 2014). It is clear from the results that with the final Viterbi implementation the best accelerations can be achieved exactly with these block sizes.

The complete execution times include the recursion and the pre- and post-processing times (e.g. initialisation and data transfer times). These figures may be informative of how much extra overhead the CUDA implementation has, such as device setup, kernel launch and host-device data transfers. When the sequential recursion time is small, the execution time of the CUDA implementation outside the recursion can become significant, but it is not expected to be an issue with realistically large datasets. Results show that the complete execution times of the CUDA implementation are somewhat stochastic and less influenced by the size of the data than the recursion times (Figure III.5), which is again realistic as most setup and post-processing overheads depend on the operating system.

The observed recursion accelerations are mostly between $\times 50$ and $\times 100$, but sometimes are approaching $\times 150$ or can be as low as $\times$1-5 (usually for extreme dataset dimensions). These speedup results can be expected to hold for sequences of any length using the streaming method (see Streaming in Section I.3.1.3 in general, or in Section 2.2.2 for the Forward algorithm) and for almost any number of inference sequences (execution can always be divided into independent runs on subsets of $\mathcal{H}'$).

Table III.4 presents quantitative runtime measurements and acceleration results from further experiments on a set of differently sized data sets. The first five rows show run time and acceleration figures for extreme datasets with minimum and maximum K, maximum L (without streaming)and with minimum and maximum N (without using virtual threads). These results are important for the assessment of performance ranges. It is clear that good performance, in general, requires the use of a high number of blocks ($=K$) and threads ($=N$) at the same time. Rows VI-VIII show results for the maximum number of threads with balanced K and L; this second set of results is especially relevant because the number of potential reference haplotypes can easily exceed 1024 in practice and the corresponding speedup results represent roughly what one may expect from the parallel implementation for such cases. Rows IX-XIV present figures for smaller reference sets and are relevant in cases where the number of reference haplotypes can be reduced in some way (e.g. by filtering according to relevance). The results suggest that runs with large K will provide the highest acceleration.

The device-host data transfer time figures correspond specifically to the device-host transfer

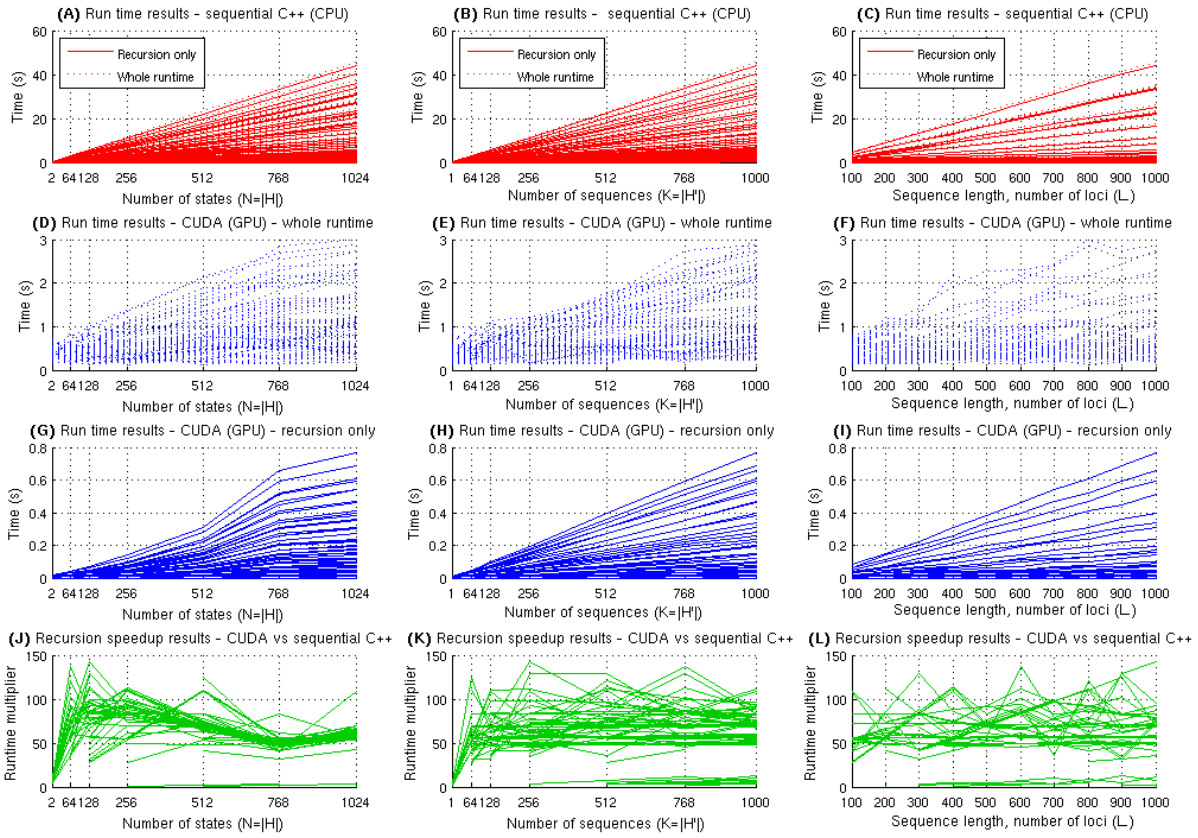

Figure III.5: Summary of recursion times, complete execution times and recursion accelerations. The first row of graphs **(A-C)** show how the complete and recursion execution times of the sequential C++ algorithm increase as a function of N, K and L, respectively. The second row **(D-F)** shows the complete execution times of the CUDA implementation, the third row **(G-I)** shows the recursion times achieved with CUDA and the last row **(J-L)** presents the calculated corresponding recursion accelerations.

of the backtrack matrix. The sum of all other data transfer times were measured to be 0-10ms in all cases (mostly 0ms) and, therefore, are deemed negligible. The data transfer times become important when assessing the balance of overlapping data transfer and kernel execution times in a possible streaming implementation. In general, the transfer times are less or about the same as the recursion times in the case of the parallel Viterbi algorithm, except for VIII, XI and XIV. The exceptions are exactly the cases when K is big and parallelisation results in the highest accelerations. Since streaming is not expected to affect the efficiency of data transfers, data transfer times are expected to be the final upper limit for performance in implementations that are aimed for use on large datasets.

Table III.4: Summary of execution time and acceleration results for the parallel Viterbi algorithm.

Dataset	Data size			Recursion times (s)		Recursion speedup	Device-host data transfer time (s)
	K	L	N	Sequential C	CUDA		
I.	1	7500	64	0.02	0.02	×1	∼0
II.	1	7500	128	0.04	0.02	×2	∼0
III.	1	7500	256	0.08	0.03	×2.6	∼0
IV.	1	7500	1024	0.33	0.05	×6.6	∼0
V.	32000	7500	2	74.70	17.49	×4.3	0.37
VI.	500	2000	1024	47.47	0.77	×62	0.78
VII.	1000	1000	1024	44.12	0.76	×58	0.79
VIII.	2000	500	1024	54.94	0.78	×70	1.10
IX.	2000	7500	64	48.87	2.06	×24	1.04
X.	1000	7500	128	46.00	2.61	×18	0.71
XI.	2000	4000	128	53.03	0.97	×55	1.10
XII.	500	7500	256	42.04	1.54	×27	0.74
XIII.	1000	4000	256	48.99	0.84	×58	0.78
XIV.	2000	2000	256	49.06	0.62	×79	1.11

$K = |\mathcal{H}'|$ is the number of inference sequences, L is the number of loci (the length of inference and reference sequences) and $N = |\mathcal{H}|$ is the number of reference sequences.

3.2.2 Empirical performance analyses for the parallel SF and FB algorithms and CCP

Since the parallel FB, SF and CCP methods were implemented for practical use, their performance was more thoroughly tested. In order to explore the behaviour of each kernel as well as the overall implementations, a larger number of experiments were conducted using various different datasets. Although empirical experiments were an integral part of development, this section presents results only for the final implementations.

Analyses of optimality

The obvious general expectation is that the performance of the parallel implementations increases with the data size. Intuitively, this is expected because large K and N mean a higher level of parallelism and large L means that the effect of overheads and unhidden data transfers become negligible. However, when data sets are really large or have extreme dimensions, various GPU and host resource constraints often alter the general trend. Further, there are some data dimensions that are 'sweet spots' for the different algorithms where performance has local peaks.

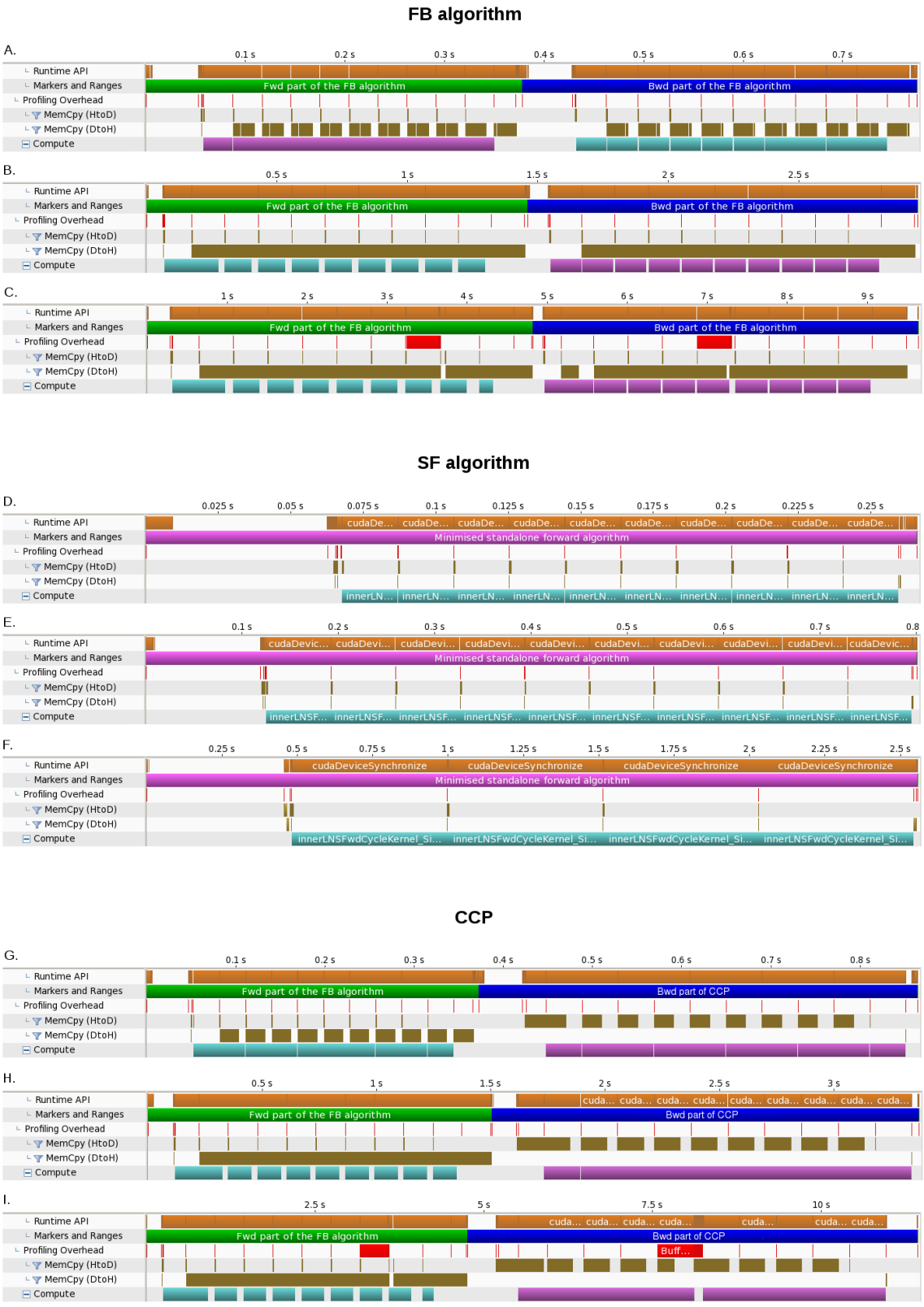

Figure III.6: Streaming dynamics analysed with the NVIDIA Visual Profiler (NVVP). The first three timelines (A-C) demonstrate streaming dynamics of the forward and backward passes of the parallel FB algorithm. The second three timelines (D-F) present dynamics for the parallel SF algorithm and the last three (G-I) present dynamics for the CCP method (from which the specialised backward pass is the one to be noted).

Since the parallel FB and SF algorithms incorporated CUDA streaming, they were optimised both at kernel level and streaming level. Streaming-level optimisations were primarily guided by analyses provided by the CUDA Visual Profiler. Figure III.6 demonstrates the typical streaming dynamics of the final implementations.

The first thing to note from the profiler results is that the parallel SF algorithm is highly kernel intensive (as expected), while data transfer and kernel execution times are relatively balanced in all FB algorithm and CCP kernels, even when a smaller number of blocks are used ($K \sim 32$) and can be almost perfectly balanced with a higher number of blocks (when $128 \leq K \leq 512$). Naturally, streaming blocks of the CCP backward pass take slightly longer than in the FB algorithm, making CCP slightly more kernel intensive.

First experiments

When analysing the performance of the parallel implementations, the first result to note is that efficiency does not change significantly with L. Beyond $L = 5000$, efficiency is not expected to increase more than 5-15% (see Figure III.7).

Low L values ($L < 5000$) effect performance only if the amount of data transfers (approximately proportional to $L_p/(L_p + L)$) not hidden by streaming becomes significant. Most analyses assume $L > 5000$, which is appropriate as most genetic data sets are expected to be much greater and $L > 5000$ may be satisfied even if analyses are split across the length of sequences.

The second important observation is that efficiency fluctuates with changes in N and K expected based on the structure of implementations. The fluctuations follow a zigzag pattern with a dampened upward trend. Efficiency peaks when the methods are run on data sets with data dimensions around $N \bmod 256 = 256$ and $K \bmod 16 = 16$ and efficiency is low when $N \bmod 256$ and/or $K \bmod 16$ are small. Fluctuations can be significant for small data sets (when $N < 2048$ and $K < 128$ simultaneously) and efficiency drops may be as large as 35%. With the increase of the number of $\mathcal{H}$ and $\mathcal{H}'$ haplotypes, fluctuations diminish and efficiency drops become less than 5%.

The zigzag pattern over N occurs because the number of CUDA threads used (the page size) is fixed irrespective of the number of virtual threads and, hence, most threads are idle in the last page when $N \mod 256$ is low, resulting in a slight performance drop. The seemingly random efficiency fluctuations, which are most apparent for the forward pass of the FB algorithm, may be due to the sensitivity of memory operations and the CUDA scheduler to the number of threads and array sizes. The reason behind the clean zigzag-patterned efficiency fluctuation over K is less certain but is most likely attributable to the granularity

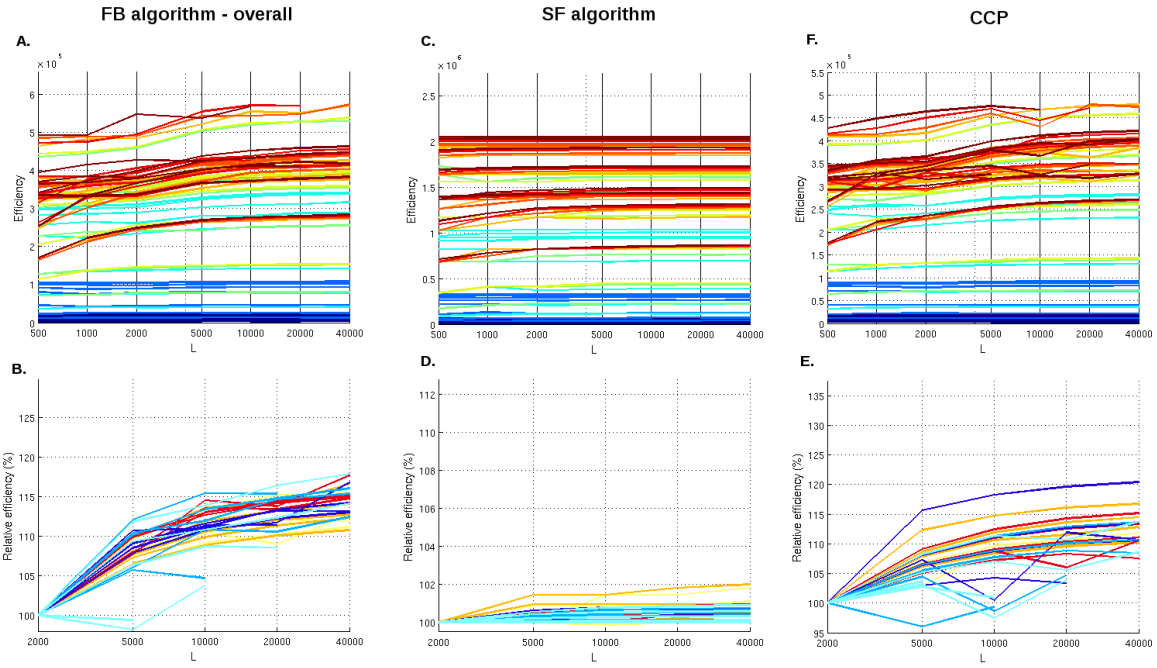

Figure III.7: Performance analysis with different sequence lengths L (L is on logarithmic scale). The first row of plots presents efficiency results across L for runs with all combinations of $K \in \{1, 5, 16, 32, 64, 128, 256, 512, 1024\}$ and $N \in \{32, 64, 128, 256, 512, 1024, 2048, 4096\}$. The second row presents efficiency changes relative to the L=2000 measurements for the more realistic size configurations with $K \in \{32, 64, 128, 256, 512, 1024\}$ and $N \in \{256, 512, 1024, 2048, 4096\}$.

of the CUDA scheduler and the assumption is that the scheduler prefers to run blocks in groups of 16 (in the case of the current four kernel implementations).

Small data sets

Efficiency is assumed to be lower for small data sets. The extent and the boundaries of efficiency drop for small datasets have been explored as follows.

A number of experiments explored changes in efficiency for various combinations of N and K (see Figure III.9). It is clear from the results that the ability of the CUDA scheduler to hide the memory latencies is crucial for performance, as the acceleration results for K=1 are very low. Part of the dependency of efficiency on N is due to the overhead of the virtual thread paging and the multi-tiered reduction methods used - both were designed for data sets with large N and represent a large overhead when N is small.

Overall, the figures show strong consistency across kernels, with mild differences regarding the optimal regions. Efficiency is generally highest for the SF algorithm and lowest for the backward pass of CCP, as expected. The kernels with more computation also reach their plateau of high efficiency faster (i.e. at smaller data dimensions), which is also expected, as

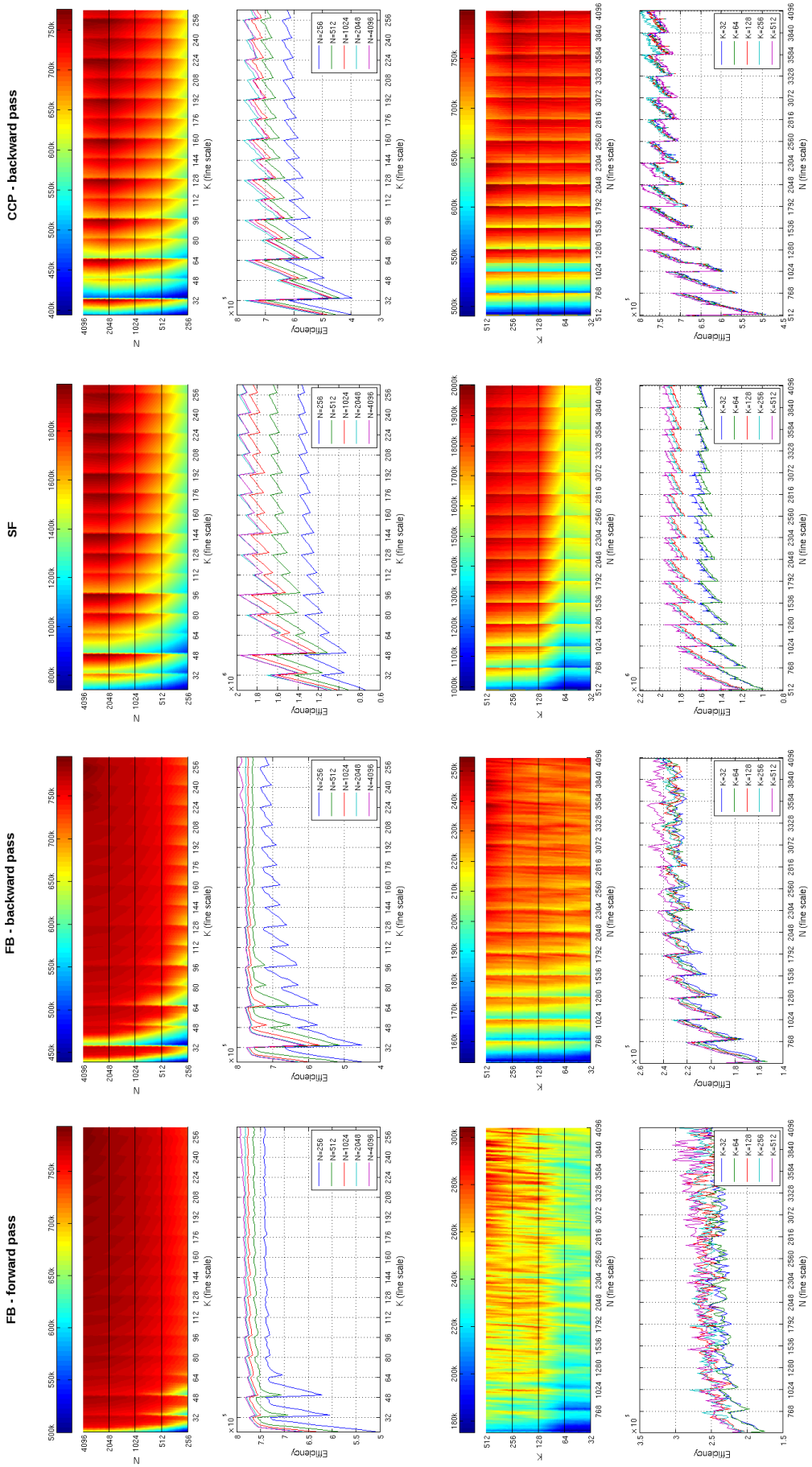

Figure III.8: Fine-scale efficiency maps - dominantly zigzag-patterned efficiency fluctuations with changes in N and K. Each coloumn of subfigures corresponds to experiments with one CUDA kernel. The top two rows of subfigures consist of corresponding efficiency heat maps and efficiency plots where the focus is on efficiency changes with N; L=2000 for the top two rows. The bottom two rows present efficiency changes with K, and L=5000.

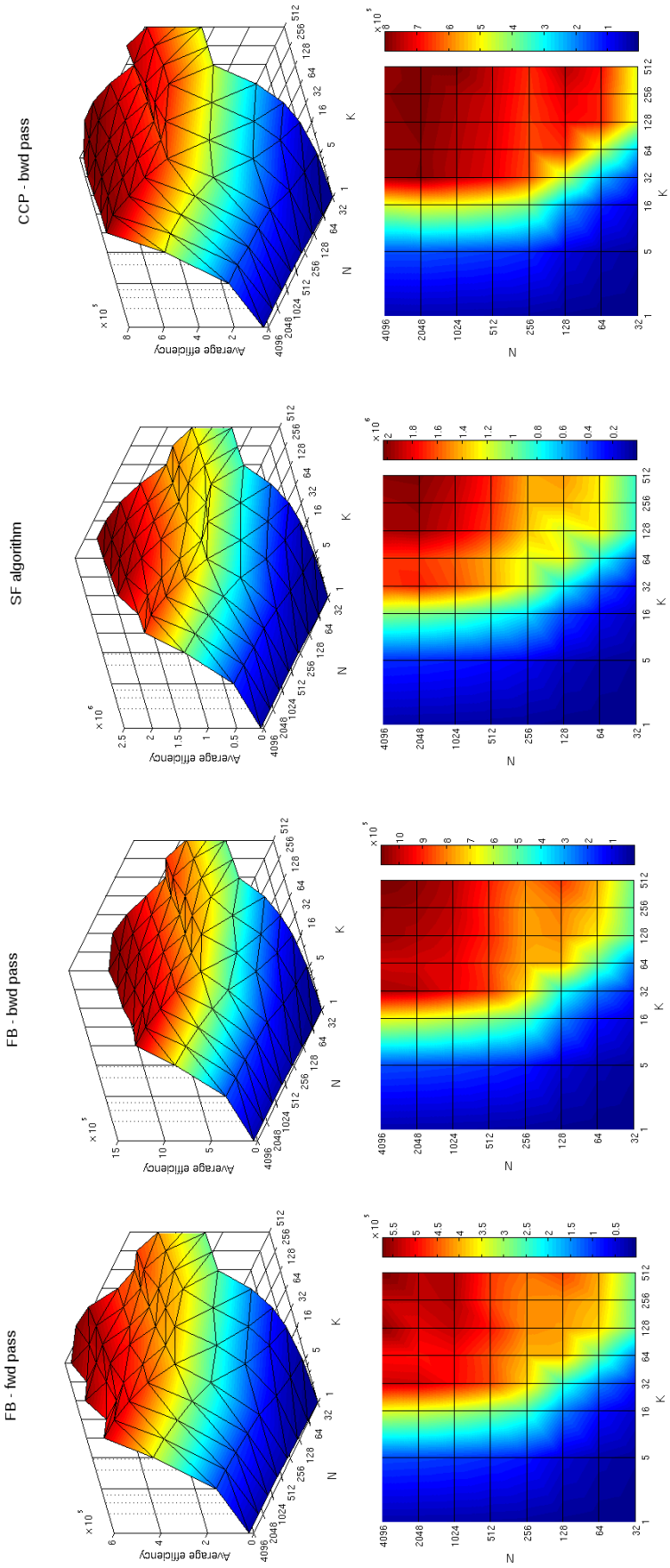

Figure III.9: Rough efficiency maps with datasets of different reference (donor) and inference (recipient) haplotype panel sizes. N and K are on logarithmic scale in all sub-figures. All datasets had L=10,000 and all N and K combinations allowed for streaming blocks of L_p =500 to fit into GMEM on the test hardware. Each coloumn corresponds to streams with a CUDA kernel. Vertically adjacent sub-figures correspond to each other - the lower ones represent a 90° elevation view.

they require lower parallelism to ensure that there is sufficient computation present to hide memory operation times.

The boundaries for the relatively flat high-performance plateaus are $K \geq 32$ and $N \geq 512$, consistently across all four kernels. The efficiency jumps sharply close to the maximum at K=32 (see Figure III.8) but increases more gradually with N.

The minimum dataset dimensions where the implementations are expected to perform well are K=32, N=128 and L=5000. However, at least K=128, $N \geq$512 and $L \geq$10,000 are required for optimal performance. These efficiency criteria are important as the efficiency of runs with datasets of smaller corresponding dimensions may be as little as 1% of the optimum (e.g. when K=1). The minimum K and N ranges correspond well to the characteristic minimum thread and block numbers expected in CUDA applications.

Naturally, in real applications the dimensions of genetic data sets are usually fixed. The size of data sets in population genetics are more than sufficient to allow for large K, N and L dimensions. However, it is possible, and sometimes necessary, to split up analyses over K (without loss of information) and along L (with potential, but possibly minimal, loss of information), mainly to be able to fit the data into host memory. The practical relevance of the recommended minimum K and L data dimension figures is that the dataset parts should not be smaller when analysis is divided.

Data sets analysed by HMM algorithms can not be divided over N without a potentially significant loss (or change) of information. In some practical situations it is possible to use a chosen subset of reference haplotypes (e.g. as shown in Howie et al. (2011)), but the performance results over N are presented mainly for assessing applicability and characterising performance.

Large data sets

In the streaming process of the FB and CCP kernels the size of streaming blocks is calculated roughly as $Lp = S^{GMEM}/12NK$ in order to be able to fit all data into device memory (S^{GMEM} is the size of GMEM in bytes). Consequently, when N and/or K are large ($N \times K >$ 0.5-1million for current GPUs with 6-12 GBs of GMEM), the streaming block size L_p can become very small ($L_p \ll 500$). Streaming is also implemented for SF, but with $Lp = S^{GMEM}/3(N+K)$, which implies that low L_p values rarely occur and are avoidable in practice, hence the SF algortihm is always suitable for large datasets with high expected performance.

In principle, very small L_p streaming block sizes could lead to reduced performance because kernel launch and memory load overheads can become significant if calculations are too few in each individual kernel run. However, if N is sufficiently large, that may counter the

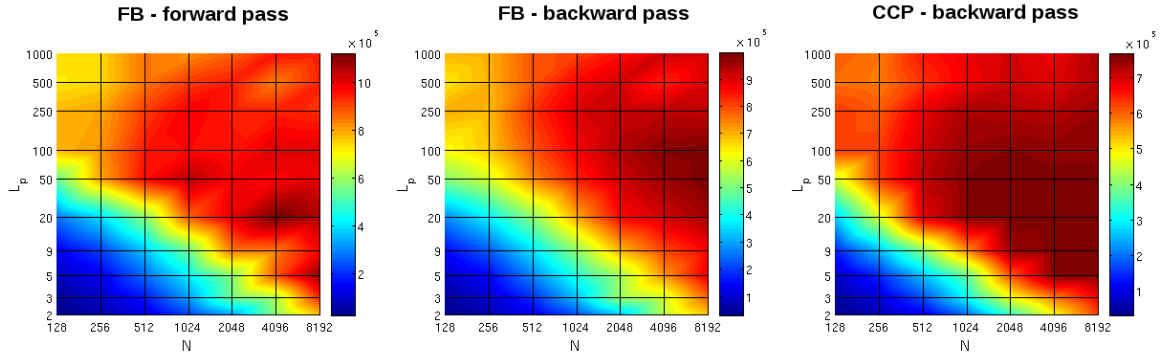

Figure III.10: Efficiency maps using different enforced L_p values. All axes are in log scale and each sub-figure corresponds to streams with a CUDA kernel. All datasets were of $K = 64$ and $L = 2000$ fixed. Only $N \in \{128, 256, 512, 1024, 2048, 4096\}$ and $L_p \in \{2, 3, 5, 9, 20, 50, 100, 250, 500, 1000\}$ were used, the rest of the heat colouring is from interpolation.

effect of small L_p streaming block sizes since each thread is expected to perform $\mathcal{O}(\, L_p \log N)$ operations in each kernel launch. Figure III.10 presents performance results for various fixed L_p streaming block sizes, where efficiency clearly depends on L_p and N jointly, thus confirming the assumptions.

The most important of the results is that even with the minimum L_p=2, efficiency approaches the maximum for all streams around N=8192. Since efficiency is expected to further improve for higher values of N, the FB and CCP methods can be expected to run with high performance when used with large datasets. For instance, assuming a GPU with only GMEM=6GBs, a grid size of K=61 can be used with datasets of N=8192, with no expected performance drop due to having a minimal L_p=2. It is also clearly observable from Figure III.10 that overall efficiency drops slightly for the FB and CCP streams when L is small and L_p is relatively large, as expected, due to the increased proportion of unhidden data transfer times. Further experiments showed (results not presented) that the inference panel size K influences efficiency independently of L_p, again, as expected.

From all performed experiments, the highest measured efficiency was noted and the upper limit of efficiency was calculated by correcting for the effects of unhidden data transfers. These results allow the estimation of relative efficiency between the different kernels for reference. Table III.5 compares the relative maximum efficiency of all CUDA streams and methods and summarises criteria for high performance (omitting the general criteria $L \gg L_p$ and the guideline of $L \geq 5000$).

The practical outcome of the above analyses is a set of guidelines for optimal use of the developed parallel methods. For instance, the approximate conditions for achieving at least

Table III.5: Summary of performance analysis experiment results for parallel implementations: approximate criteria for high performance and (lower) estimates of highest achievable efficiency.

Stream	Approx. criteria for ≥80% efficiency				Highest efficiency estimate		
	$N \times L_p$	K	N	$N \times K$	Eff. est.* (1/ms)	Observed for dataset	L_p
FB - fwd	≥ 25k	≥ 32	≥128	≥ 16k	1,318,882	K=256, N=8192	211
FB - bwd	≥ 80k	≥ 32	≥128	≥ 8k	1,103,696	K=1024, N=2048	212
FB - full	**≥ 80k**	**≥ 32**	**≥ 128**	**≥ 16k**	**602,473**	**K=512, N=4096**	**211**
SF - full	≥ 7k	**≥ 32**	**≥ 128**	**≥ 65k**	**2,285,041**	**K=4096, N=65536**	**500**
CCP - bwd	≥ 12k	≥ 32	≥ 64	≥ 8k	818,373	K=512, N=2048	210
CCP - full	**≥ 12k**	≥ 32	**≥ 128**	**≥ 16k**	**497,697**	**K=512, N=2048**	**210**

* Efficiency is calculated as $E = KLN/T_{execution}$ as defined earlier from empirical measurements. The upper limit of efficiency is estimated as $E^* = E(L+L_p)/L$.

$K = |\mathcal{H}'|$ is the number of inference sequences, L is the number of loci (the length of inference and reference sequences) and $N = |\mathcal{H}|$ is the number of reference sequences.

80% of the maxmimum efficiency would be $N \times L_p \geq$12,000 for CCP, $N \times L_p \geq$25,000 for the forward pass and $N \times L_p \geq$80,000 for the backward pass of the parallel FB algorithm.

Streaming-level investigations of optimality

Results from the Visual Profiler (see Figure III.6) present typical dynamics for the recommended minimum K=32, the adequate K=128 and the presumably optimal K=512. The transfer of the forward and backward result tables overwhelmingly dominates device-host communication and is responsible for most of the HtoD and DtoH transfer times. Since the result tables are stored as single-precision floats, the observed transfer times are likely very close to the minimum achievable transfer times (with the current version of PCIe).

The kernel intensity of the SF algorithm (Figure III.6, timelines D-F) means that the execution times (hence the efficiency and accelerations) are dictated only by the efficiency of the SF kernel. Consequently, in the case of the SF algorithm, further kernel optimisations could theoretically improve overall performance.

The relative balance of the FB and CCP streams (see timelines A-C, G-I) suggests that their implementation is around an (at least local) optimum. It is safe to assume that under current (Fermi) architectures, further significant performance improvements could be achieved only if both data transfers and kernel run times were improved.

3.2.3 Acceleration results for the parallel SF and FB algorithms

After isolated parallel performance experiments, a set of empirical execution time experiments were performed to compare corresponding sequential C++ and CUDA implementations. Table III.6 presents the measured execution time, calculated efficiency and acceleration results from these experiments. The efficiency of the parallel implementations ranges between 2% and 93% of the estimated relevant maxima. Parallel acceleration ranges between ×29 and ×90 for the FB algorithm and between ×9 and ×50 for the SF algorithm. These acceleration results are determined by a multitude of factors on both the sequential and parallel sides. The efficiency of sequential implementations is generally higher for smaller datasets, while the efficiency of CUDA implementations is higher for large data sets. The consequent disparity results in the observed relatively wide acceleration spectrum. Nevertheless, all acceleration results are in the range that one might expect when a CUDA implementation is compared to relatively optimised sequential implementations.

The first group of experiments (I-VII) presents results for the minimum suggested K=32, the second set (VIII-XIV) presents results for the presumed optimal K=512 and the last set (XV-XXIII) demonstrates the change of efficiency and accelerations with the change of K.

Despite the use of double-precision floating point numbers, virtual threads and scaling in the FB algorithm, acceleration results appear slightly higher than in the case of the parallel Viterbi algorithm (see Table III.4 for the Viterbi results). This may be attributable to the higher level of general code optimisation of the parallel FB algorithm. The lower accelerations achieved for the SF algorithm result from the simplicity and significantly lower amount of memory operations of the sequential SF algorithm, which allow it to be relatively efficient as a sequential implementation (about ×10 as efficient as the sequential FB algorithm).

3.2.4 The performance of CUDA-Chromopainter

The performance of CCP was also evaluated through a series of experiments. The execution time of chunk count calculations was compared against a minimised version of the OCP (MCP), a purpose-built sequential implementation of Chromopainter and the original (OCP). MCP was created by eliminating all instructions from the OCP source code that were not essential for calculating chunk counts. The OCP and MCP methods were also run with minimum settings (0 EM iterations, 0 samples and minimal output).

The results are summarised in Table III.7, from which it is clear that the accelerations achieved by CCP are quite significant. The measured ×250-×800 accelerations can reduce execution times from days to a few minutes and performance is expected to be around the upper end in the case of large datasets, which the methods are designed for.

Table III.6: Comparison of performance results between the parallel and sequential SF and FB algorithms.

Experiment			FB algorithm results					SF algorithm results				
			Sequential		CUDA			Sequential		CUDA		
Dataset	K	N	t(s)	E	t(ms)	E	X	t(ms)	E	t(ms)	E	X
I.	32	128	2.4	8.5k	83	247k	×**29**	242	85k	27	759k	×**9**
II.	32	256	4.8	8.6k	108	379k	×**44**	494	83k	35	1,170k	×**14**
III.	32	512	9.8	8.4k	185	443k	×**53**	1,050	78k	57	1,437k	×**18**
IV.	32	1,024	19.7	8.3k	343	478k	×**57**	2,236	73k	102	1,606k	×**22**
V.	32	2,048	40.1	8.2k	652	503k	×**62**	4,615	71k	192	1,707k	×**24**
VI.	32	4,096	81.9	8.0k	1,305	502k	×**63**	9,932	66k	396	1,655k	×**25**
VII.	32	8,192	178	7.4k	2,656	493k	×**67**	20,632	64k	790	1,659k	×**26**
VIII.	512	128	38.4	8.5k	744	440k	×**52**	3,934	83k	224	1,463k	×**18**
IX.	512	256	76.0	8.6k	1,565	419k	×**49**	7,798	84k	463	1,415k	×**17**
X.	512	512	156	8.4k	2,739	479k	×**57**	16,788	78k	770	1,702k	×**22**
XI.	512	1,024	344	7.6k	5,339	491k	×**64**	40,838	64k	1,374	1,908k	×**30**
XII.	512	2,048	734	7.1k	10,399	504k	×**71**	93,354	56k	2,606	2,012k	×**36**
XIII.	512	4,096	1,652	6.3k	19,191	546k	×**86**	216,622	48k	5,233	2,004k	×**41**
XIV.	512	8,192	3,247	6.5k	37,361	561k	×**87**	482,476	43k	10,331	2,030k	×**47**
XV.	1	2,048	1.3	8.2k	485	21k	×**3**	152	67k	164	62k	×**1**
XVI.	5	2,048	6.3	8.1k	488	105k	×**13**	754	68k	164	312k	×**5**
XVII.	16	2,048	20.4	8.0k	505	324k	×**40**	2,426	68k	164	999k	×**15**
XVIII.	32	2,048	40.1	8.2k	652	503k	×**62**	4,615	71k	192	1,707k	×**24**
XIX.	64	2,048	80.4	8.1k	1,309	501k	×**61**	9,236	71k	394	1,663k	×**23**
XX.	128	2,048	165	7.9k	2,528	518k	×**65**	19,034	69k	671	1,953k	×**28**
XXI.	256	2,048	365	7.2k	4,958	529k	×**74**	44,446	59k	1,328	1,974k	×**33**
XXII.	512	2,048	734	7.1k	10,399	504k	×**71**	93,354	56k	2,606	2,012k	×**36**
XXIII.	1,024	2,048	1,502	7.0k	18,771	559k	×**80**	193,392	54k	5,162	2,031k	×**37**

In the dataset definitions $K = |\mathcal{H}'|$ is the number of inference sequences, $N = |\mathcal{H}|$ is the number of reference sequences. The length of inference and reference sequences (L) was chosen to be 5,000 for all data sets due to memory constraints (and not because it is the expected sequence length in population genetic datasets).

In the results sections t refers to execution time - the unit for sequential FB is 1s, whereas it is 1ms for all other methods. $E = KLN/t$ is the perceived efficiency and is not adjusted according to L and L_p. $X = t^{seq}/t^{CUDA}$ is the magnitude of parallel acceleration.

All results are based on average execution times from more than 10 experiment repetitions.

It is also clear that the acceleration figures with respect to OCP are a result of an interplay of different contributing factors. One such factor is that OCP performs substantial calculations that are not essential for finding chunk counts. With the absence of these calculations, the minimised OCP was about ×2 times faster. It is also clear that the implementation of the OCP is far from optimal - a reasonably optimised sequential version that was implemented specifically for comparison purposes achieved about a further ×3 higher speed. Finally, the effect of parallelisation can account for about ×40-×90 acceleration.

Table III.7: Summary of performance results for CUDA-Chromopainter (CCP) and the parallel SF algorithm; comparisons with a sequential C++ implementation, the original Chromopainter (OCP) and a minimized version of OCP (MCP).

	Experiment				CCP/SF		OCP		MCP		Seq.C++	
	Dataset	K	N	L	t(s)	E	t(min)	X	t(min)	X'	t(min)	X''
Chromopainter calculations (with CCP)												
I.	1kG-CEU	170	170	151,627	16.4	267k	78.1	×**285**	35.4	×129	12.6	×46
II.	1kG-YRI	176	176	231,996	25.8	278k	127.7	×**297**	53.8	×125	21.0	×49
III.	1kG-ALL A	2,184	2,184	2,500	27.0	442k	363.3	×**807**	97.3	×216	37.7	×84
IV.	1kG-ALL B	30	2,184	365,644	52.7	454k	500.5	×**569**	221.7	×252	71.7	×82
V.	1kG-ALL C	32	2,184	182,822	25.7	498k	256.0	×**598**	113.4	×265	38.1	×89
Likelihood calculation (with parallel SF)												
VI.	1kG-CEU	170	170	151,627	4.1	1,077k	78.1	×**1,151**	35.4	×522	1.03	×15
VII.	1kG-YRI	176	176	231,996	6.1	1,186k	127.7	×**1,265**	53.8	×533	1.74	×17
VIII.	1kG-ALL A	2,184	2,184	2,500	6.3	1,906k	363.3	×**3,484**	97.3	×933	4.27	×41
IX	1kG-ALL B	30	2,184	365,644	16.9	1,417k	500.5	×**1,776**	221.7	×787	6.38	×23
X	1kG-ALL C	32	2,184	182,822	8.5	1,507k	256.0	×**1,812**	113.4	×803	3.33	×24
XI.	1kG-ALL full	2,184	2,184	365,644	**946.8**	1,842k	np*	n/a	np*	n/a	np*	n/a

* 'np' means that experimental runs on this dataset were not performed due to resource and time constraints.

$K = |\mathcal{H}'|$ is the number of inference sequences, L is the number of loci (the length of inference and reference sequences) and $N = |\mathcal{H}|$ is the number of reference sequences

All experiments were performed on haplotypes of chromosome 22 from the 1000 Genomes project. CEU refers to a subset of the 1000 Genomes data with European ancestry and YRI refers to haplotypes with African (Yoruban) ancestry. ALL refers to the whole 1000 Genomes sample set, and 'ALL A', 'ALL B' and 'ALL C' refer to different partitions of the complete set (in order to be able to load it into host memory).

'MCP' is a minimised version of OCP such that operations and instructions not essential to chunk count calculations are removed. 'Seq.C++' refers to own, adequately optimised sequential c++ implementations of the SF algorithm and CCP (in their respective table sections).

$E = KLN/t$ is the perceived efficiency and is not adjusted according to L and L_p.

$X = t^{OCP}/t^{CCP}$, $X' = t^{M.OCP}/t^{CCP}$ and $X'' = t^{SEQC}/t^{CCP}$ are the magnitudes of acceleration achieved by the parallel implementation with respect to the sequential methods.

Likelihood calculation

The original Chromopainter was not designed for efficiently finding the likelihood, but the purpose of the built-in EM iterations is to find the MLE of various model parameters.

From a practical point of view, it is worth noting that the parallel SF algorithm executes ×1100-×3500 faster than a single EM iteration of OCP. While the two methods are not exact substitutes to each other, the three magnitudes of speed difference and the fact that the parallel SF algorithm is applicable to large datasets with ease while OCP is not, suggests that the parallel SF algorithm is much better suited for exploring the parameter space. Naturally,

a very efficient sequential SF algorithm implementation under the LS model could also be applicable for parameter space explorations, but not on the large scale that the parallel SF allows (the speed difference between sequential and parallel is expected to be ×20 to ×40-fold).

3.3 Application results

3.3.1 CUDA Chromopainter (CCP)

One core application of the parallel FB algorithm was CCP. The key results of this application are that CCP is able to handle large datasets (as explained in Section 2.2.2 and Tables III.2 and III.3) and that it achieves accelerations of ×250-×800 on practical datasets compared to the original (see Section 3.2.4) while it is highly accurate and stable (as demonstrated in Section 3.1.2),.

3.3.2 Mapping the likelihood in the $M_g \times N_e$ parameter space

The parallel SF algorithm was applied to calculate the joint log-likelihood for the CEU, YRI and the whole 1000 Genomes set at given sets of (N_e, M_g) grid points. The grid for the CEU (N^{CEU}=K^{CEU}=170) and YRI (N^{YRI}=K^{YRI}=176) sets was set as $M_g \in [0,\ 0.008]$ and $N_e \in [0,\ 110k]$, with 101 points in each coordinate (about 10,000 evaluations in total). Using all non-singleton variants (L^{CEU}=151,627 and L^{YRI}=231,996), the complete analysis took 11.6 and 17.6 hours, respectively. Since the complete 1000 Genomes Project set is much larger (N^{ALL}=K^{ALL}=2,184 and L^{ALL}=365,644), only a 28×36 grid was calculated in the region $M_g \in [0,\ 0.0013]$ and $N_e \in [150k,\ 750k]$ and the map was split into 24 regions so that likelihood calculations were distributed to 24 GPUs . The whole parallel calculation (1008 evaluations) took 11.4 hours to complete.

The grids were refined by standard (cubic) interpolation algorithms and the (N_e, M_g) pairs for the highest interpolated log-likelihood were taken as the MLE. The grid for the complete 1000 Genomes set is rough, but given the relatively smooth and predictable log-likelihood surface makes the interpolation reliable and, hence, the MLE can be expected to be reasonably accurate.

The likelihood maps and MLEs found with the parallel SF algorithm and the EM traces of the OCP are presented in Figure III.11. The definition of the recombination scaler N_e followed that of the original LS model $N_e^{LS} = \frac{N_e^{OCP}(N+K)}{4}$.

As expected, N_e was estimated to be higher for the YRI set than for the CEU set, but it

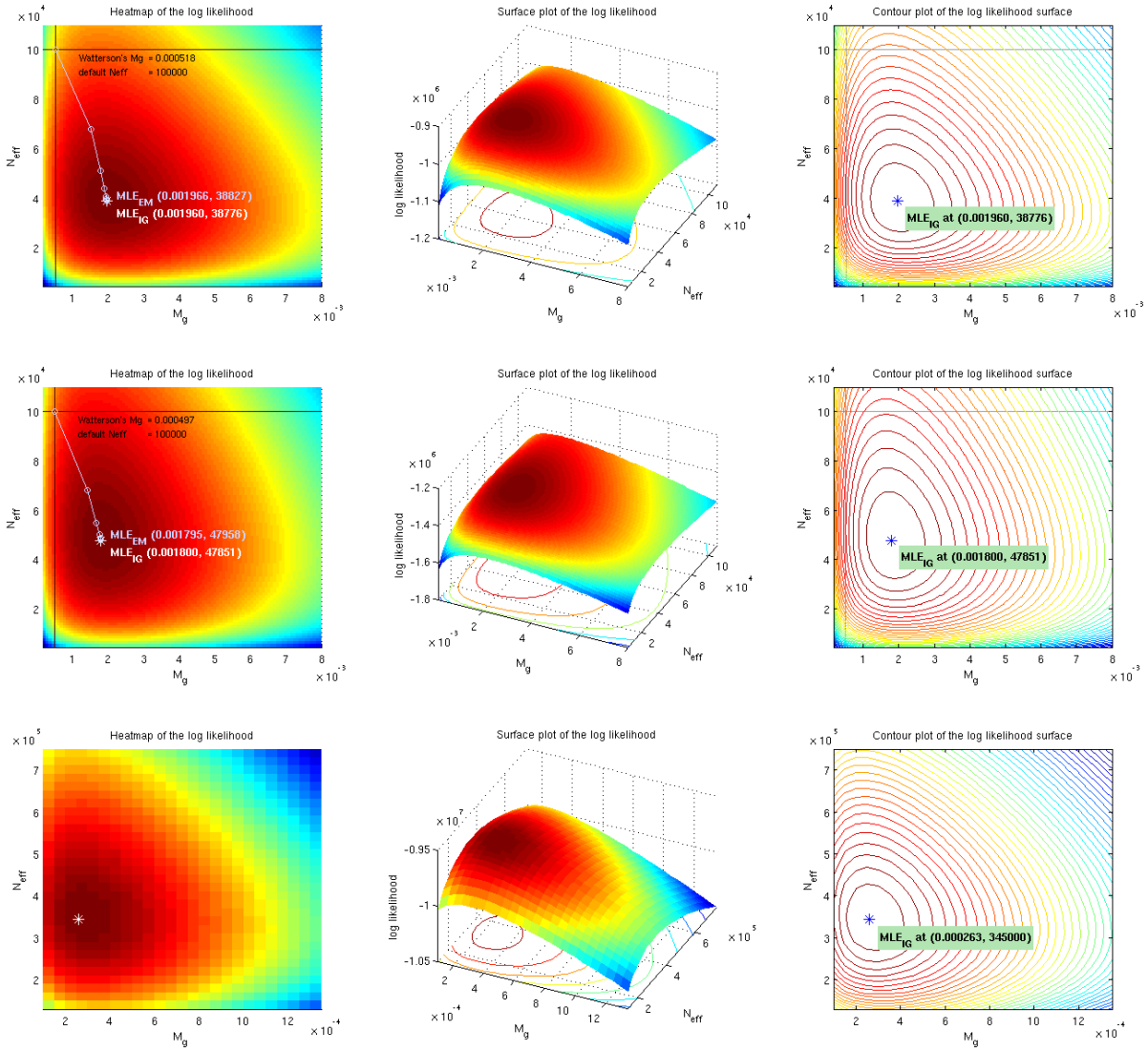

Figure III.11: Parameter space explorations under the Li and Stephens model: maps of the joint log likelihood surface based on haplotype sets on chromosome 22 of CEU (top) and YRI (middle) and all (bottom) populations from the 1000 Genomes Project. The top two rows represent results from a 100×100 grid of joint likelihood values. The bottom row is based on a 28×37 grid. The grids in the first column are shown without alteration. The middle column clarifies how the colors translate into log-likelihood values and the last column represents contour plots after interpolation. For each dataset the MLE based on the maximum of the interpolated grids is shown as MLE_{IG}. For the CEU and YRI sets, the EM algorithm-based MLE estimates of OCP are plotted for the first ten iterations for comparison. The MLE at the last (10^{th}) iterations is marked as MLE_{EM}. The black lines correspond to the default values in the OCP - Watterson's estimate for M_g and 100k for N_e (as it is back-transformed to present the MLE originally defined in the LS model).

is striking that the MLEs found by the parallel SF algorithm for the whole 1000 Genomes Project set (N_e^{ALL}=345k, M_g^{ALL}=0.00026) are significantly different from those in the YRI set (N_e^{YRI}=47,851, M_g^{YRI}=0.0018).

The MLE results from the OCP (the 10^{th}, final value given by the EM algorithm) and the interpolated grid maxima are closely agreeing in the case of both the CEU and the YRI sets. In fact, the EM algorithm of the OCP seems to have converged close to the final result in only four iterations. The OCP was not run on the complete 1000 Genomes dataset due to resource constraints, hence there is no trace for EM iteration results in the corresponding plots.

3.3.3 Screening datasets based on MLEs for N_e, M_g and the log-likelihood

When producing the likelihood maps with the grid-based approach in Section 3.3.2, the population log-likelihoods were produced from the individual haplotype likelihood results. In the next application, the individual haplotype log-likelihoods were used directly. With haplotype results for all grid points, the MLE was approximated after interpolation for each haplotype, separately. A 100×100 grid was used in the $M_g \in [0,\ 0.008]$ and $N_e \in [0,\ 110k]$ space for the CEU and YRI and 28×36 grid in the $M_g \in [0,\ 0.0013] \times N_e \in [150k,\ 750k]$ space for the complete 1000 Genomes set.

The distribution of the MLEs for N_e, M_g, the maximum log-likelihoods and a first analysis of their correlation are presented in Figure III.12. The limited boundaries of the grid did not allow to find all MLEs correctly - some outliers are expected to have MLEs beyond the grid. Nevertheless, most values can be assumed valid and relatively accurate.

The results indicate that there is a very strong correlation between M_g and the likelihood, as it may be expected - the R^2 values are 0.96, 0.9, 0.76 for the CEU, YRI and the complete 1000 Genomes dataset, respectively. There is also a strong correlation between N_e and the likelihood, with respective R^2 values of 0.40, 0.69, 0.72. Naturally, the correlation between M_g and N_e is also apparent, but it is considerably lower.

It is striking that there are some obvious outliers or outlier groups in each dataset. As a large M_g is indicative of sequencing errors, the MLE for M_g may be useful as a criteria for filtering datasets before using them in downstream analysis.

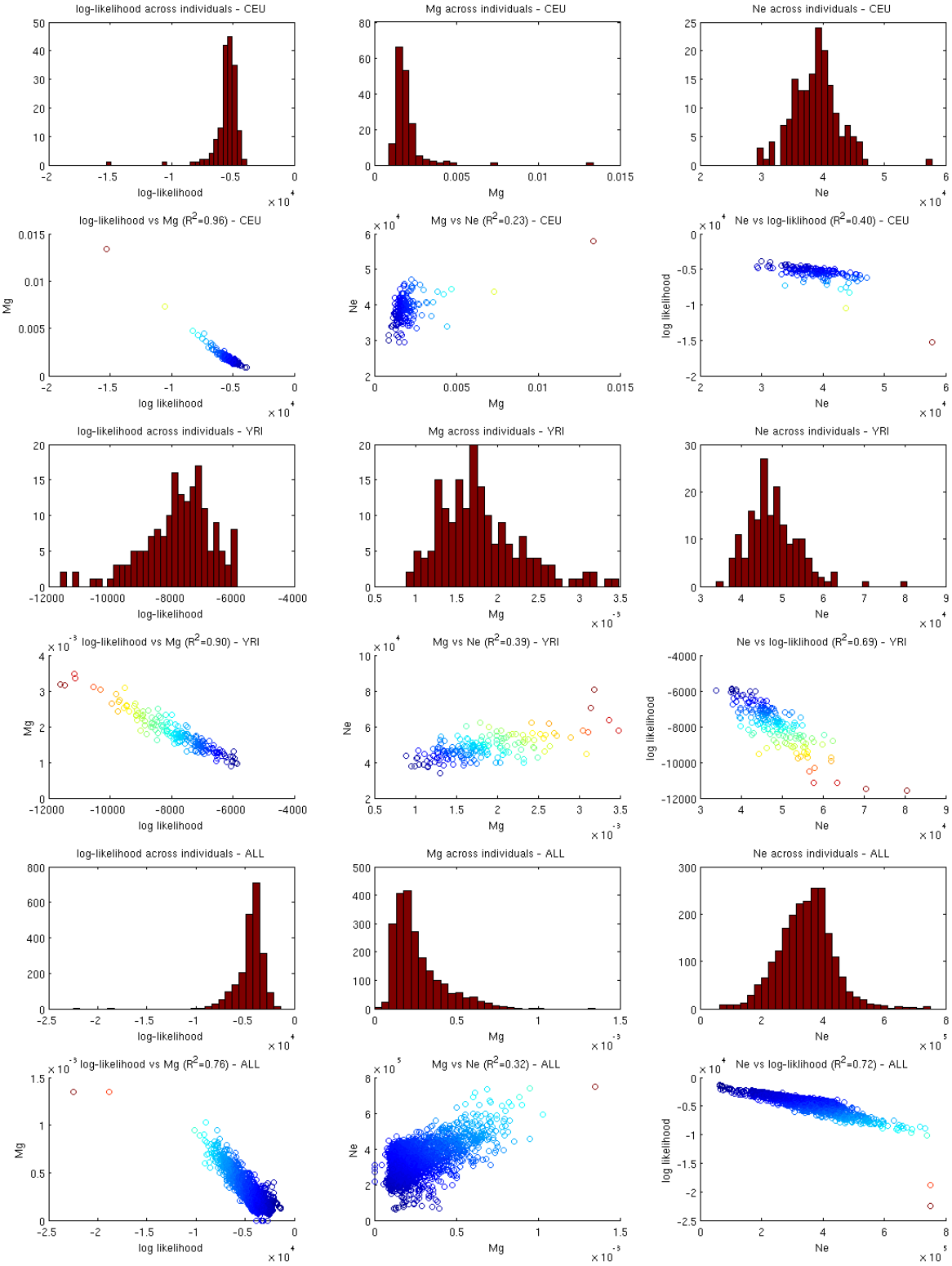

Figure III.12: Exploring the distribution of N_e, M_g and the log likelihood at the MLE across haplotypes from various populations from the 1000 Genomes Project. All plots correspond to analyses on chromosome 22 haplotypes. The first two rows of plots present results on the CEU set, the third and fourth rows correspond to the YRI set, and the last two rows present results on the whole 1000 Genomes set. The colouring of the scatterplots is according to the likelihood, which is intended to clarify the middle scatter plots.

4 Discussion and conclusions

4.1 Parallelisation under the LS model and the practicalities of using GPUs

Standard HMM algorithms are highly suitable for parallelisation as has been demonstrated in previous works (Cartey et al., 2012; Du et al., 2010; Horn et al., 2005; Hymel, 2011; Li et al., 2009; Liu, 2009; Nielsen and Sand, 2011; Sand et al., 2010; Turin, 1998; Zhang et al., 2009) and shown in Sections 2.1.2 and 2.2.1. It has also been shown that efficient communication between parallel threads is crucial in order to achieve performance when parallelising over the states. Since inter-thread communication is most efficient using GPGPUs among the presently available parallel computing architectures, they have been the focus through most of this chapter.

Sections 2.1.2 and 2.2.1 further demonstrate that parallelisation is also possible under the LS model. This is particularly useful as datasets in population genetics tend to be large and performance improvements from parallelism over a large number of inference sequences and states is highly valuable. The large dataset dimensions allow us to use a large number of CUDA blocks and threads in GPUs, which allows us to utilise their capabilities of hiding various costs of memory access, such as latency and bandwidth limitations. It is also demonstrated that with appropriate algorithm design and implementation structure (such as CUDA streaming and virtual threads), it is possible to avoid most limitations of GPUs that may be relevant when using large datasets, such as limited device memory (GMEM).

With the effective elimination of GPU memory limitations, the strongest limitation that remains is the host memory. Even computers that are set up for scientific computing tend to have insufficient operative memory for storing results from HMM algorithm on a genome-wide scale. However, it is possible to overcome these limitations by dividing datasets (and hence analysis) over observations without losing information (and accuracy of results) or even over sequence lengths (with potential minor information loss). This chapter offers ample information on how it is best to divide datasets to retain the maximum possible performance with the current implementations. Correct dataset division is crucial not only when one aims to work with limited host memory, but also when analysis is divided to be performed across multiple GPUs, which is particularly promising for use in population genetics.

As a special case, parallelisation of the Forward algorithm is especially convenient when results are required only for the last locus (α_{s_L}). The reduction of memory requirements allows the parallel implementation of the Forward algorithm to be readily applicable even

with architectures that have limited host memory.

The various validation results presented in this chapter prove the accuracy, stability and robustness achievable with CUDA implementations of parallel HMM algorithms under the LS model. The derived tools, CCP and likelihood mapping have also been proven accurate.

4.2 Discussion of results

Performance results

In terms of execution time performance, thorough experimentation showed that large data sets allow for better overall computational efficiency for the implemented parallel methods, contrary to what is experienced with sequential implementations where efficiency drops with increased problem size. The consequence is that the achieved practical parallel acceleration is significant - empirical results show that they can reach up to hundreds and thousands for large datasets (up to $\times 80$ for the Viterbi, $\times 90$ for the FB, $\times 45$ for the SF, $\times 800$ for CCP and up to $\times 40$-$\times 3500$ for large-scale likelihood calculation).

Performance experiments were thorough but not completely exhaustive and the true highest and lowest efficiency and acceleration figures may be slightly different from the figures presented, but that should not diminish the value of the results. The accelerations are considerable on their own, especially given that the implementations were designed to handle large data sets as a priority (which limited optimisations) and that all acceleration figures presented in this chapter resulted from comparisons with adequately optimised sequential implementations and the original Chromopainter.

For comparison, Table III.8 presents acceleration results with parallel HMM algorithms in previous works. When comparing results, it is important to note that results in these works were achieved with implementations of general HMMs that are somewhat easier to parallelise efficiently compared to those under the LS model. Further, the extent of optimisation in the various GPU and CPU implementations is often not explained and not clear from these works, hence the acceleration results may occasionally be overstated and should be observed and assessed with caution. Despite the complication in parallelisation difficulty and irrespective of the potential limitations of results in past work, the parallel accelerations presented in this chapter are clearly competitive.

The practical significance of the achieved accelerations can not be overstated. Such increases in speed can lead to a great improvement in the applicability of existing methods. The convenience alone that results from speed can allow researchers to be more efficient in their work, repeat analyses with large data sets, experiment with different parameterisations, explore

Table III.8: HMM algorithm speedups achieved on GPUs in past work

Tool/method	HMM algorithm(s)	Problem or application area	Acceleration achieved	Comparison benchmark	Citation
HMMingbird	All	Profile HMMs	30×-103×	HMMoC (seq., optimised) (Lunter, 2007b)	Cartey et al. (2012)
HMMingbird	All	Occasionally dishonest casino	21×-50×	HMMoC (seq., optimised) (Lunter, 2007b)	Cartey et al. (2012)
cuHMM	Forward	General HMMs, no assumptions	70×-880×	Own sequential C code, not optimised	Liu (2009)
cuHMM	Baum-Welch	General HMMs, no assumptions	60×-200×	Own sequential C code, not optimised	Liu (2009)
-	Fwd-bwd	Random generated HMMs	up to 25×	Own sequential C code	Li et al. (2009)
-	Viterbi	Keyword spotting	2.9×-3.2×	HVite (parallel)	Zhang et al. (2009)
Streaming	Viterbi	Sequence alignment	0.85×-5×	Own sequential	Du et al. (2010)
Tile-based	Viterbi	Sequence alignment	2×-6×	Own sequential	Du et al. (2010)
-	Forward	dummy test model	up to 181 ×	Own sequential C code	Hymel (2011)
-	Viterbi	dummy test model	up to 4.6 ×	Own sequential C code	Hymel (2011)
-	Baum-Welch	dummy test model	up to 65 ×	Own sequential C code	Hymel (2011)

The acceleration figures correspond to the implementations achieved in the corresponding work in the area or on the data specified under the 'Problem or application area' column compared to the 'Comparison benchmark' method.

multiple analysis scenarios and compare results on differently filtered/processed data sets, which are all essential in population genetics. With the increase of data sizes - soon expected to be in the hundred thousands and millions of genome-wide samples - low-performance methods will become inadequate very quickly.

While the parallel SF algorithm may not offer as high an acceleration as the Viterbi and FB

algorithms and CCP would (compared to sequential implementations), its extreme practical speed and convenient applicability make it the most readily applicable method presented in this chapter.

Application results

CUDA Chromopainter

The first application of parallel HMM models under the Li and Stephens model was Chromosome painting. For this application, the principal outcome is the accelerated version, CUDA-Chromopainter, which produces accurate and reliable results in 1/250-1/800 time (Section 3.2.4).

Likelihood mapping and population parameter estimation

The second application was likelihood mapping and population parameter estimation with the repetitive use of the SF algorithm (see results in Section 3.3.2). Visualisation of the likelihood surface is a result that was previously impossible to obtain. The smoothness of the likelihood maps and the MLE results from grid interpolation follow expectations and agree with the EM-found estimates of OCP in the case of the CEU and YRI data sets. In fact, the EM algorithm of the OCP has found relatively accurate results in just four EM iterations. While it is impossible to prove that the likelihood surface is unimodal based purely on the grid results, the agreement between the two approaches is strong indication that the results of the EM algorithm in the OCP can be trusted.

However, the unexpected MLEs found for the complete 1000 Genomes dataset may shed light on unexpected and potentially alarming behaviour of the LS model in large-scale analyses. First, the M_g being significantly different across the different sets does not seem very plausible (M_g^{ALL}=0.00026 compared to M_g^{YRI}=0.0018). Second, while the MLE N_e difference between the YRI and the whole datasets is expected, the extent of difference may not be (N_e^{ALL}=345k compared to $N_e^{YRI}=47,851$). The YRI population is thought to capture a large portion of the ancestral haplotypes, hence $N_e^{YRI} \ll N_e^{CEU}$ is not expected.

One possible explanation for the unexpected N_e^{ALL} and M_g^{ALL} results follows from the fact that increasing the sample size increases the chance of finding individuals with locally identical sequences. If we see the LS model as a method that explores relatedness only at the tips of ancestral recombination graphs, it is clear that such relatedness can be explained fully using locally identical sequences that may occur more frequently in large samples. Accordingly, it is natural that the MLE of N_e falls in the high ranges and consequently the MLE of M_g is small.

It is possible that in the presence of a large number of haplotypes, the LS model overwhelmingly explains variation with the recombination-like mosaic structure and much less with mutations/errors. Possibly coincidentally, the factor of difference observed for the N_e and M_g parameters between the YRI and complete 1000 Genomes set are each other's relatively accurate inverse (7.2 and 1/6.9 between N_e and M_g estimates, respectively).

It is also clear from the results that the default parameters of the OCP are very different from the actual maximums. It has been argued regarding multiple methods based on the LS model (e.g. Chromopainter and IMPUTE2) that parameterisation, especially that of N_e is not required to be precise in order to produce accurate inference results. Although parameterisation may not be crucial for most applications, the validity of results may be questionable with random parameters while trusted parameterisation should improve confidence in results (or their absence should prompt analysis and call for proofs on why parameterisation does not matter in the LS model).

Given the importance and wide use of the LS model, the unexpected results call for further investigation regarding the behaviour of the LS model with large samples. The CUDA implementations of HMM algorithms presented in this chapter (and/or future derivatives) may be key tools in this direction of research.

Dataset screening with parameter estimation for individual haplotypes

The estimation of various parameters has been possible in the past with approaches such as the EM algorithm used in OCP. However, the parallel SF algorithm allows analysis to be performed on a much greater scale and much more practically.

The distributions of parameter MLEs for individual haplotypes suggest that some haplotypes have strikingly high M_e estimates and low likelihoods in the same time. These results may suggest potential large-scale experimental errors for these haplotypes. Further analysis regarding the exact reasons for the existence of outliers may be required, but the efficient exploration of the parameter distributions is essential and is made possible with the parallel SF algorithm. The significance of dataset screening comes from the sensitivity that is expected in population genetics in downstream analysis.

The dataset screening scheme presented for the global (sequence-wide constant) mutation rate and recombination scaler are just the first obvious applications of parameter estimation-based screening. With some alterations of the parallel SF algorithm it may be possible to estimate mutation rates and recombination scalers for each locus. With distributions of parameters over and across sequences it may be possible to uncover phasing errors (switch errors) or local sequencing errors by developing a scheme for characterising certainty of certain data points based on parameter estimates.

4.3 Limitations and future work

Possible future performance improvements and analyses

A possible final limitation of the acceleration results is that the current comparison does not include parallel CPU implementations. While the GPU acceleration results compared to sequential code are significant in their own right, further comparison with a parallel CPU implementation could be performed if one is interested in investigating what the highest achievable acceleration may be across all architectures. The principal possibility for future work in this regard could be to include an optimised multi-threaded parallel CPU implementation in the performance comparisons.

Possibilities for further model simplifications and CUDA optimisations could also be investigated and analysed. One possible model simplification would be the use of constant recombination rates. Making the model homogeneous would reduce some memory requirements, constant cache misses and the number of indexing operations, which could all result in some efficiency improvements. Unfortunately, the resulting speedup would be traded for the accuracy of results, which is why this scheme has not been explored in this work.

Optimisations have been explored on a wide scale for most implementations. It is very likely that more efficient implementations could be achieved with further effort, but most further optimisations are not expected to have significant effect. In the case of the FB algorithm and CCP both kernels and data transfers would need to be optimised. In the case of the SF algorithm there may be possibilities for considerable improvement from further optimising the corresponding kernel, but given the minimalistic implementation of the parallel SF algorithm, there is not much room for change.

These observations and analyses assumed current GPU technology (specifically NVIDIA Fermi architectures). The data transfer vs. kernel execution time balance may be very different for future GPUs. Future improvements of the CUDA platform may also offer new optimisations and/or reduced communication costs, which may tip the balance of the current implementations either way. For instance, a possibility with the Kepler architecture is that warp shuffles may significantly improve the efficiency ($\sim$60%) of reduction operations due to hardware support, which would then improve the performance of kernels and streams (up to 20% for FB and up to 40% for SF and CCP based on experiment-based estimates - data not presented).

Limitations, possible improvements and extensions for implementations

Further development for the Viterbi algorithm: Since the parallel Viterbi algorithm was implemented for exploration purposes, it lacks the ability to handle large datasets, which could be allowed by implementing streaming and the use of virtual threads as for the parallel FB, SF and CCP methods.

Host memory limitations: The main applicability limitations of the parallel FB and CCP methods result from expected host memory sizes. Although large memory architectures (with ∼1TB of operative memory) exist, currently most compute servers are expected to have 16-96GBs of host memory (at least per node), which is sufficient for storing only parts of the HMM result matrices in realistic datasets (see Table III.3). The only way for using the parallel FB and CCP methods with really large datasets is by dividing analyses over observation sets (without loss of information) or over the length of sequences (with potential loss of information). The first division is trivial and advisable, however it should be applied keeping in mind that $K \geq 32$ should be preserved for each subset in order to retain high performance. In case a dataset with $K = 32$ does not fit into memory, the division over L may be performed. While the results will potentially be slightly inaccurate at division points, it is important to point out that this approach has been widely used in population genetics. A method called 'Chromocombine' has been developed for Chromopainter to reassemble results from parts of sequences and the method IMPUTE2 is advised to be performed on few MB sections of sequences with overlaps.

Dataset partitioning: As dataset partitioning may be necessary but inconvenient in most analyses (especially over L), a useful convenience extension could be to automatically partition given data sets over L and perform analysis with these partitions consecutively (similarly to streaming, but now host-level code would be overlapped with data loads from HDD/SSD). Beyond the obvious convenience improvements, the advantages of doing the partitioning in code is that the optimal partitioning may be performed over K and L from an efficiency point of view. Further, integrated partitioning could allow for partition over sequence length without loss of information as Viterbi, forward and backward pass results may be transferred from one partition to the next automatically. The downside of automatic partitioning is that bandwidth for data transfers between storage and host memory become the main limitation - in such cases loads from HDD can be expected to dominate runtime and negate performance, while loads from SSD may be fast enough to potentially be on par with computation speed.

Automatic distribution to compute nodes: Perhaps the most useful way of integrating auto-partitioning would be if the method also distributed the workload to multiple GPUs and across multiple different nodes. Such an extension would be highly useful for conveniently using the algorithm on multi-GPU systems, but the task distributions can also be performed using scripts with relative ease and, hence, this extension was not in the scope of this work.

GMEM limitations: The second most prominent limitation for the parallel FB and CCP methods and the primary one for the parallel SF algorithm is the device memory available for GPUs. Although the soft limits due to GMEM (see Table III.3) are not expected to be relevant for some time, it is reasonable to expect that the device memory of GPUs will increase in the coming years, hence allowing the SF algorithm to run with datasets of N=65535 and K=65535 and the FB and CCP algorithms with the largest datasets that the host memory size allows. It may also be possible to moderately reduce the GMEM requirements by optimising streaming for space (a possible 33% reduction) and using single-precision floats where it doesn't reduce accuracy (a potential 0%-50% reduction in space requirements). In the case that GMEM becomes the limiting factor, dataset partitioning over the observation set may be applied with ease and without any expected performance loss. It can also be expected that in future parallel architectures the accelerators (GPUs or other specialised processors) are expected to have closer connection with CPUs and device memory may be merged with host memory. Naturally, such architectures will require complete re-adaptation of the parallel implementations.

Datasets beyond 65,535: The current implementations are not expected to be stable with datasets where $N \geq$65,535 or $K \geq$65,535. When running such large jobs becomes relevant (especially when N is large) in the future, the implementations will require thorough robustness analysis in order to work reliably on different systems.

Removing limits on K: The limit ($K \leq$7186) on the observed sequences in CCP is set in order to keep P(D) in constant memory. This limit may be completely eliminated by moving P(D) out of CMEM and, hence, slightly reducing performance. This scheme was not implemented because under expected host memory limits datasets are expected to be divided over the observation sequences and datasets. In the future, when the limit on K becomes relevant, the optimizations may be restructured according to the technology available at the time.

Auto-tuning: The implemented CUDA kernels rely on the use of a fixed page size *TMAX* for virtual threads and a very simple approximation of the optimal streaming block size L_p. It may slightly improve efficiency and could ensure more robust performance if $TMAX$ and L_p were set dynamically considering all possible factors (at least the dataset dimensions). Potentially a useful option could be to find the optimum values by empirical measurements on a few mini trial runs before large analyses.

Extensions for CCP: The current version of CUDA-Chromopainter works with a subset of the options that the OCP allows. In case of sufficient demand, functionality for the further options may be implemented, such as more versatile handing of input datasets, handling diplotypes and extra outputs. CCP also does not calculate chunk lengths and doesn't perform sampling, which could be added. With some modifications, it may also be possible to implement the EM algorithm for parameter estimation in a parallel fashion,

which could have great applicability for dataset screening.

Directions for future research

The most important direction for fundamental research is to thoroughly investigate the behaviour of the LS model with large sample sizes.

The most useful practical direction would be to implement a parallel version of the EM algorithm for fast MLE approximation for population level dataset screening and to explore possibilities for uncovering errors in the dataset across loci individually.

It may provide new insights regarding the error structure of data from the HapMap and the 1000 Genomes projects if various population-wise and individual MEL results were compared between the two.

Further work may also include further applications, such as the use of the parallel FB algorithm to produce a parallel version of GeneCluster (Su, 2008). As the lstree program appears to be very slow for the purpose of similarity matrix generation, it would be a potential direct application of a CUDA and could also improve the run time performance of the single-locus variant calling method in Chapter II.

4.4 Conclusion

The most direct aim of this chapter is to demonstrate the importance of the algorithmic and computational aspects of parallel method development in population genetics through a case study. The parallelisation of the HMM methods and the corresponding analyses are presented in significant detail to provide a reference point for the parallel development of methods in the future. The hope is that this work will inspire research in parallel method development in genetics and will serve as a starting point and/or a worked example for how to go about it.

This work is also the first on exploring the possibilities for HMM parallelisation under the LS model. It is argued and demonstrated that parallelisation under the LS model requires domain-specific development. The extent of achievable accelerations has been explored with practical CUDA implementations; the accuracy and stability of these implementations has been proven and their limitations investigated.

Consequently, direct outcomes of this work are a readily usable set of high-performance methods for the FB and SF algorithms, a high-performance CUDA implementation of the Chromopainter method as well as a simple method to efficiently map the log-likelihood un-

der the LS model for populations of haplotypes. The FB, SF and CCP methods achieve significant performance improvements that can greatly improve the quality of research (and potentially lower the costs) when they are applied. Likelihood map results offer novel insights into the workings of the LS model and also properties of large population genetic datasets. The likelihood calculation-based applications suggest multiple interesting avenues for research, such as analysing the behaviour of the LS model on large datasets, or the development of various dataset filtering algorithms.

Perhaps the most important message of this chapter is that parallel thinking and high-performance implementations are very important in population genetics (also generally in computational statistics and scientific computing on the whole). Researchers often understand that the applicability of methods is greatly influenced by their performance, but what is less obvious is that extreme performance can also make existing methods applicable to new purposes, or methods previously not deemed practical to be revisited.

Naturally, algorithmic efficiency should always be the first goal in performance-purposed method development but in the case of algorithms with polynomial complexity, parallelisation should be preferred to trading performance for accuracy with approximations. Keeping this principle in mind, the development of (statistical or other) methods should always be attempted with the potential of parallelisation. In fact, the best approach is to consider parallelisation from the outset in order to avoid missing any opportunities. When such thinking directs research, it can reform the way methods and theory is developed - towards more practically, more widely applicable and potentially more diverse tools.

Chapter IV

BLOCK-PARALLELISM FOR HIDDEN MARKOV MODELS

1 Introduction

The core of standard Hidden Markov Model (HMM) algorithms are dynamic programming recursions. These recursion often contribute to the majority of computation requirements in HMM-based methods. Improvements in the execution time of these recursions can directly affect application performance and the overall applicability of virtually every method where HMMs are used. As discussed in Chapters I and III, there are multiple options for parallelising these recursions. One can run calculations in parallel for different models (parametrisations) or for multiple observations with ease due to the independence of calculations and when the number of models or observations is sufficiently large, these approaches should be an adequate first choice. Beyond such trivial parallelisms, one can parallelise the dynamic programming recursions in HMM algorithms over the present states and previous states (see Chapter III for a specific application under the Li and Stephens model).

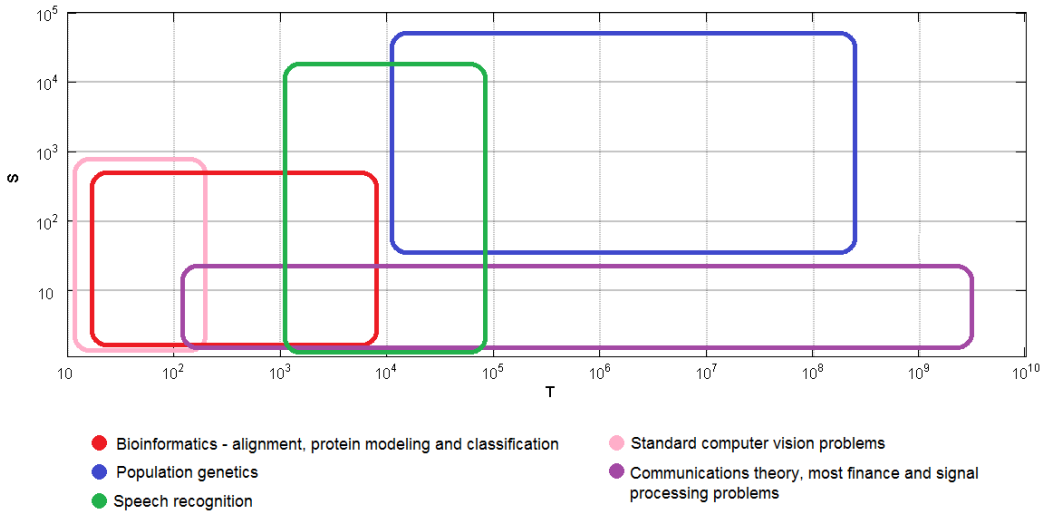

Figure IV.1: Typical problem sizes - observation lengths (T) and numbers of hidden states (S) - for various HMM application areas.

In many HMM applications, however, the length of observations (T) is orders of magnitude greater than the number of observations (K) and the number of hidden states (S). Figure IV.1 shows where typical methods in various application areas fall in terms of problem size.

For instance, in certain areas of genetics, finance, signal processing (e.g. classification of time-series data) and speech recognition, the number of states may be well below hundred, while observation lengths can be in the thousands or even in the millions. In such applications, parallelising the dynamic programming recursions of HMM algorithms over the length of observations may allow for greater parallelisms compared to standard approaches over K and S, which otherwise may not even be sufficient to utilise the parallelism allowed by the computing platforms available.

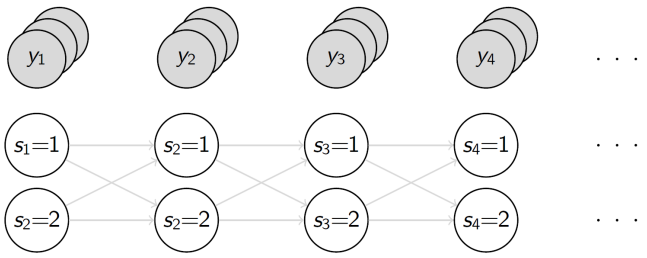

Figure IV.2: A typical trellis for HMM algorithms with S=2 states and K=3 observation sequences of length T>4. The 3D structure of HMM dynamic programming 'tables' suggests three possible approaches for parallelising HMM algorithms.

In other cases, the available computation platform may be segmented in a way that the efficiency of communication between certain blocks of processing units (and, hence, certain parallel threads) is insufficient for the effective cooperation of parallel threads because communication times outweigh processing time. In these scenarios, parallelisms over S are often not applicable, and those over parametrisations (G parametrisations) and K may be insufficient to utilise the available processing power.

The subject of this chapter is the parallelisation of dynamic programming recursions over blocks of the observation length in HMM algorithms. This will be called HMM block-parallelism (BP). Prior works in BP are discussed in Section 2, a probabilistic explanation of BP is presented in Section 3, and Section 4 explores options for HMM parallelisation (these will be called strategies) in terms of their applicability and relative performance. It will be shown that BP is relevant for most important problem sizes, it will be shown that it can be effective on distributed computing platforms and that it may be optimal on standard parallel platforms and GPU-based systems when S is not too large.

1.1 Block-parallelism for HMMs

The fact that the dynamic programming tables of HMM algorithms are three-dimensional ($K \times S \times T$, see Figure IV.2), suggests the possibility of parallelisms over T (naturally, there would be a separate dynamic programming table for every parametrisation). While paral-

lelism over parametrisations G and K are trivial in every respect[1] and parallelisms over S require only a suitable computational platform, the possibility for parallelism over the length of observations is not straightforward, even on a theoretical level.

The complexity comes from the sequential dependence of the dynamic programming calculations, i.e. calculations at t depend on results from calculations at time $t-1$. The consequence of this dependency is that concatenating outputs from independent HMM algorithm runs on consecutive blocks of observations generally does not lead to correct results (see Figure IV.3). In order to apply block-parallelism, it is imperative to be able to perform calculations independently on each block in a way that correct results can be acquired at the end of the calculations in some way.

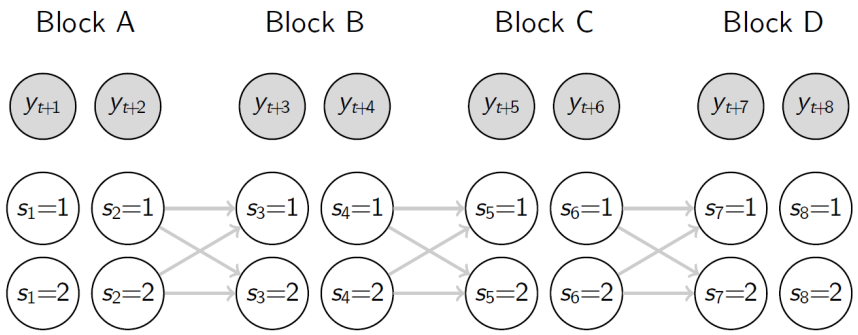

Figure IV.3: Parallelisation over blocks of observation sequences with one observation sequence of T=8 pieces of observations and S=2 states. In this partition, there are four blocks of length two (B=4, T_b=2).

Additionally, as it will be shown later in this chapter, the resource (computation and space) requirements of block-parallelism often scale faster with the number of states compared to standard sequential and parallel HMM algorithms (see Sections 2 and 3).

The consequence is that the applicability and relevance of block-parallelism is strictly limited to scenarios where $S \ll T$ and the level of parallelism achievable on the computation platform (C) is also high ($S < C$ at least). This relatively high complexity, its high C requirement and its somewhat limited applicability may be the reasons why this direction of HMM parallelism has not received sufficient attention in the past. Given the recent rapid improvement of available parallel platforms, block-parallelism may be a relevant approach for many applications in the close future, especially when combined with the more traditional parallelisation approaches (see Section 4 for theoretical performance results).

[1] Development is trivial, there are no requirements for the parallel computation platform (can work with distributed systems) and there are no computation or performance costs introduced apart from the result aggregation at the end of each complete run

Assumptions

In all analyses in this chapter, standard general HMMs are assumed with a discrete, finite and known set $\mathcal{S}$ of states, a given transition matrix A, given discrete emission probability distributions E, and a given initial distribution π. It is also assumed that $\boldsymbol{A}$ and $\boldsymbol{E}$ are fixed but the discussed methods can also be trivially applied to inhomogeneous HMMs where the parameters are time-dependent (with $\boldsymbol{A^{(t)}}$ and $\boldsymbol{E^{(t)}}$). There are no assumptions about the observations apart from the standard requirement that all observations come from a discrete emission alphabet.

The objective is to reduce the execution times (ϱ) of the dynamic programming recursions for the forward, the forward-backward and the Viterbi algorithms by running calculations in parallel on blocks of observations (see Figure IV.3). Since the maximum parallelism limit C and communication costs Q will naturally affect the applicability and optimality of the block-parallelisation strategies used, performance has to be analysed in the light of these parameters.

While the order of space requirements are given for all parallel algorithms, it is not in the scope of this work to consider the space requirements nor to propose parallel algorithms optimised for minimum space requirements. It is clear that the space requirements will be an important decision factor in the practical adoption of block-parallelisation - there is no doubt that it may be improved in future works for cases where it is necessary. For now, it is assumed in performance analyses that the available memory space is infinite.

It is further assumed for simplicity that there is a single observation sequence of length T and a single parametrisation considered[2]. The observation ($\boldsymbol{y} = \{y_1, ..., y_T\}$) is dissected into n disjoint 'calculation sections', each of length $L = T/n \quad (L, T, n \in \mathbb{N})$. Calculations corresponding to different calculation sections are performed sequentially.

Each calculation section is also partitioned into B disjoint blocks of length $L_b = L/B \quad (B, L_b \in \mathbb{N})$ and calculations corresponding to different blocks is performed in parallel in every calculation section when BP is applied[3].

[2]While the $K = 1$ assumption should not limit the validity of the derivations and analyses in this chapter, it is important to keep in mind that in practice it is often beneficial to include parallelism over K an G besides parallelism over states and observation blocks when applicable.

[3]The assumptions $L \mid T$ and $B \mid L$ should also not limit the validity of derivations and assessments. When these assumptions do not hold, it is still possible that performance is unaffected in practice. Alternatively, there may be a slight performance loss, but it is not expected to be comparable with the extent of acceleration achievable by parallelisation, except when problem sizes are overly small for parallelisation in the first place.

'Algorithm classes' and 'strategy types'

While the asymptotic performance characteristics of the dynamic programming recursions in different HMM algorithms are identical, it will be necessary to distinguish two classes of HMM algorithms based on a more subtle level of their complexity. Class I consists of algorithms where results are calculated only for a very few time points. A characteristic example is the standalone forward algorithm, where $P(s_T \,|\, \mathbf{y})$ are calculated only for $t=T$. Class II consists of algorithms where results are calculated for most $t \in T$, including the forward-backward, Viterbi and posterior state path sampling procedures.

For all HMM algorithms, it is possible to apply different parallelisms and even to try multiple combinations of them. These combinations will be termed parallelisation 'strategies' in this chapter. It is expected that there will be a qualitative difference between strategies where $L_b=1$ is fixed and strategies where parallelisation is performed using longer blocks ($L_b>1$). These two assumptions on L_b separate strategies into two distinct types.

1.2 Notation

- For simplicity, $K=1$ and $G=1$ are assumed and the number of sequences (K), the observation sequence indices $k \in \{1,..,K\}$, the number of parametrisations (G) and the parametrisation indices $h \in \{1,..,G\}$ are omitted from most notation.
- T is the length of the observation and $t \in \{1,..,T\}$ is the running index over positions of the observation sequence.
- L is the length of the section of observation ('calculation section') that calculations are performed on at a time. The total observation length T is assumed to be dissected into L-long calculation sections ($L \leq T$). For sections beyond the first, the initial messages (or the initial state distributions) can be acquired from the results on the preceding sections. The case $L \leq T/2$ is useful mainly for allowing extra flexibility in parallel optimisations (but may also be useful in practice to limit space requirements in certain use cases). For simplicity, uniform section lengths and $L \mid T$ are assumed. The number of calculation sections is $n=T/L$. In derivations and analyses where L is not relevant, $L=T$ will be assumed for simplicity.
- $\boldsymbol{y} = \{y_1,..,y_T\}$ is the observation sequence vector, where y_t denotes a piece of observation at time t. Similarly, $y_{t_m:t_n}$ refers to a sub-sequence vector $\{y_{t_m},..,y_{t_n}\}$, e.g. $y_{1:10}$ refers to the first 10 observations.
- $\mathcal{S} = \{1,..,S\}$ denotes the set of HMM states, with $i,j \in \mathcal{S}$ as state values. Further, s_t refers to the state corresponding to the observation y_t at position t and $s_{t_m:t_n}$ refers to the state sequence vector $\{s_{t_m},..,s_{t_n}\}$.

- The HMM parameters are defined as $\boldsymbol{\lambda}=(\boldsymbol{A},\boldsymbol{E},\boldsymbol{\pi})$. Here $\boldsymbol{A}$ is the transition matrix $\boldsymbol{A}=\{a_{ij}=\mathrm{P}(s_{t+1}=j|s_t=i)\}$, the emission probability distributions are $\boldsymbol{E}_j=\{e_j=\mathrm{P}(y_t|s_t=j)\}$. The initial distribution is given as $\boldsymbol{\pi}=\{\pi_j=\mathrm{P}(s_1=j)\}$. Where applicable, the number of parametrisations will be G and the running index over parametrisations will be g. For simplicity, $\boldsymbol{\lambda}$ is often omitted from most notation.

- $\mathcal{B}=\{1,..,B\}$ is the set of observation blocks defined by a given partition with B blocks and $b\in\mathcal{B}$ is the running index over these blocks. For simplicity, uniform partitions and $B\mid L$ are assumed.

- L_b denotes the length of block b in the current partition. Assuming $B\mid L$, it is trivial that $L_b=L/B$ and $\sum_{\mathcal{B}}L_b=L$. Block-parallel strategies can be divided into two types based on whether they assume $L_b=1$ or $L_b>1$ (the two types are not to be confused with the two algorithm classes defined below).

- τ_b and τ'_b are the time steps corresponding to the starting and the end points of block b, respectively. Here $\tau_b=\left(\sum_{m=1}^{b-1}L_m\right)+1$ and $\tau'_b=\sum_{m=1}^{b}L_m$. For uniform partitions, $\tau_b=(b-1)\,L/B+1$ and $\tau'_b=bL/B$.

- The observation sub-sequence corresponding to block b is $\boldsymbol{y}_b=\{y_{\tau_b:\tau'_b}\}$, where $|\boldsymbol{y}_b|=L_b$. Further, the set of corresponding time points are $\boldsymbol{t}_b=\{\tau_b,..,\tau'_b\}$.

- $\mathcal{R}_b=\{1,..,S\}$ is the set of S 'conditional sub-runs' ('sub-runs' for short) in block b and $r\in\mathcal{R}_b$ is the running index over sub-runs.

- $\alpha(s_t)$, $\beta(s_t)$ and $\gamma(s_t)$ are the respective recursion terms for the forward and backward passes of the forward-backward algorithm and the forward pass of the Viterbi algorithm. The corresponding terms that condition on the state at the end of the previous block $b-1$ (or next block $b+1$) are $\alpha_r^{(b)}(s_t|s_{\tau'_{b-1}}=r)$, $\beta_r^{(b)}(s_t|s_{\tau'_{b+1}}=r)$ and $\gamma_r^{(b)}(s_t|s_{\tau'_{b-1}}=r)$.

- C is a parameter of the available parallel computation platform and represents the highest level of effective practical parallelism achievable in practice. The value C can be thought of as the upper limit of parallelisations, defined by the available technology. For generality, it may be useful to think of $C_{eff}=C_{max}/K$ when parallelism over multiple sequences is also to be applied ($C=C_{eff}$ will be assumed for simplicity, unless otherwise stated). Unfortunately, it is usually not trivial in practice to determine what the highest degree of achievable parallelism may be, as it depends not only on the application and the type (e.g. cloud/distributed, CPU, GPU, FPGA, etc.) and specific instance of hardware used, but also on the implementation platform (e.g. CUDA, OpenCL, MPI, OpenMP, OpenACC, HDL, etc.), the specific structure of implementations, memory use, applied optimisations, compilers and the run-time environments (e.g. operating system, concurrently running applications, etc.).

- $Q^{(platform)}$ is introduced in this chapter (Section 4.1.1) as a further parameter of parallel computing platforms under the simplified communications model (SCM). This parameter reflects the efficiency of communications and synchronisations allowed by the platform and is assessed qualitatively, using four general grades $Q \in \{0, L, M, H\}$, which stand for 'effectively zero', 'low', 'medium' and 'high', respectively. In the case of strategies, $Q^{(W)}$ refers to the communication and synchronisation requirements of strategy W.
- $D^{(W)}$ is the highest (theoretically) achievable parallelism for the parallel strategy W assuming $C = \infty$. The parametric formula for $D^{(W)}$ depends only on the design of the parallel strategy, while its actual 'value' is expected to depend on the problem size (T and S) and on the parallelisation parameters B and L as well. $D^{(W*)}$ denotes the algorithmically achievable highest parallelism with optimal parameters (B^* and L^*, which usually depend on S and T and are expected to be different across strategies).
- $P^{(W)}$ refers to the extent of absolute maximum achievable effective parallelism $P^{(W)} = \min\{D^{(W)}, C\}$, with strategy W and given computing power C.
- Z_1, Z_2 and Z_3 refer to various added computational costs incurred when parallelising HMM algorithms. Z_1 is the direct computation cost incurred by parallelising the recursions, Z_2 refers to the amount of computation required by the merge steps (in BP algorithms) and Z_3 refers to the computation costs incurred when parallelising the merge steps.
- $\rho^{(W)}$ refers to the algorithmic lower bound of parallel runtime for strategy W, and is calculated using D, while $\varrho^{(W)}$ refers to the constrained parallel runtime and is calculated using P. The $\rho^{(seq)} = \varrho^{(seq)}$ denote the sequential runtime.
- $\varsigma^{(W)}$ is the space requirement for parallel strategy W.
- $\chi^{(W)} = \varrho^{(seq)}/\varrho^{(W)}$ is the parallel acceleration achievable with algorithm W ($\chi^{(W)}$ is given as a speed multiplier), and $\xi^{(W)} = (\chi^{(W)}/D^{(W)})$ is the efficiency of strategy W.

2 Prior Works

In prior works, most of the effort in HMM block-parallelism was based on matrix formulations of the dynamic programming recursions (Nielsen and Sand, 2011; Turin, 1998).

Some solutions have been published for parallelising the forward and backward recursions and the Viterbi and Baum-Welch algorithms but with minimal or no probabilistic explanation. There has also been no works exploring possible combinations of parallelisms[4] and their

[4]Combinations of parallelisms are called 'strategies' in later sections of this chapter.

optimality, and performance evaluations of parallel HMM algorithms have also been minimal in prior works.

2.1 Matrix forward and backward algorithms

Foundations - matrix probability and recursions in matrix form

Turin (1998) focused on accelerating the complete Baum-Welch algorithm, but also provided solutions for parallelising the forward and backward recursions with various basic parallelisation strategies. Turin demonstrated that the forward and backward recursions can be expressed in matrix form, by defining the 'matrix probability'

$$\mathbf{P}(y_t) = \mathbf{A}\mathbf{E}^{\mathbf{d}}(y_t) \tag{IV.1}$$

corresponding to a piece of observation y_t, where $\mathbf{A}$ is the transition matrix and $\mathbf{E}^{\mathbf{d}} = \text{diag}\{e_j = \text{P}(y_t|s_t = j\}$ is a diagonal matrix of the emission probabilities. Following this definition, the probability $\text{P}(\mathbf{y}|\boldsymbol{\lambda})$ of a sequence $\mathbf{y} = \{y_1, y_2, ..., y_T\}$ is

$$\mathbf{P}(\mathbf{y}) = \boldsymbol{\pi}\ \mathbf{P}(y_1)\ \mathbf{P}(y_2)\ \cdots\ \mathbf{P}(y_T)\ \mathbf{1} = \boldsymbol{\pi}\Big(\prod_{t=1}^{T}\mathbf{P}(y_t)\Big)\ \mathbf{1}, \tag{IV.2a}$$

where $\mathbf{1}$ is a column vector of ones. It is easy to see that eq. (IV.2a) is a restructured form of the standard forward algorithm, and as Turin highlighted, the alpha messages can be calculated in a 'matrix forward algorithm' as

$$\boldsymbol{\alpha}_1 \ = \ \{\alpha(s_1)\} \ = \ \boldsymbol{\pi}\ \mathbf{P}(y_1), \tag{IV.3a}$$

$$\boldsymbol{\alpha}_t \ = \ \{\alpha(s_t)\} \ = \ \boldsymbol{\alpha}_{t-1}\ \mathbf{P}(y_t) = \boldsymbol{\pi}\ \prod {}^{T}_{t=1}\mathbf{P}(y_t). \tag{IV.3b}$$

Similarly, the backward pass of the forward-backward algorithm may be performed as a 'matrix backward algorithm' as

$$\boldsymbol{\beta}_T \ = \ \{\beta(s_T)\} \ = \ \mathbf{P}(y_T)\ \mathbf{1}, \tag{IV.4}$$

$$\boldsymbol{\beta}_t \ = \ \{\beta(s_t)\} \ = \ \mathbf{P}(y_t)\ \boldsymbol{\beta}_{t+1} = \Big(\prod {}^{T}_{t=1}\mathbf{P}(y_t)\Big)\ \mathbf{1}. \tag{IV.5}$$

Strategies and performance

Turin observed that matrix probabilities $\mathbf{P}(y_{t_a:t_b})$ corresponding to blocks of observations $y_{t_a:t_b}$ can be performed in parallel by independently calculating each of the $\mathbf{P}(y_1), ..., \mathbf{P}(y_T)$ and then obtaining the block-wise results by matrix-matrix multiplications $\mathbf{P}(y_{t_a:t_b}) = \prod_{t=a}^{b} \mathbf{P}(y_t)$. These operations can be performed independently for every block, and the block-wise results can later be merged.

Turin showed that the merge can be performed in a tree-reduction form, since the order of calculations are not important in eq. (IV.2a). In a tree-reduction-based merge, a binary tree dictates the order of matrix-matrix multiplications, with each node corresponding to results on a subset of consecutive blocks of the observations (on the leaf level, a node corresponds to a single block). Results corresponding to a parent node ($\mathbf{P}(y_{t_a:t_c})$) are calculated by multiplying the results of its children as

$$\mathbf{P}(y_{t_a:t_c}) = \mathbf{P}(y_{t_a:t_b})\, \mathbf{P}(y_{t_b+1:t_c}). \tag{IV.6}$$

Turin also pointed out that a sequential merge of results on consecutive blocks (called the forward-only and backward-only algorithms in his work) may have better performance. In sequential merge, $\boldsymbol{\pi}\mathbf{P}(y_1)$ is calculated in the first step (resulting in a vector), and $\boldsymbol{\pi}\ \mathbf{P}(y_{1:k-1})\ \mathbf{P}(y_k)$ in the k^{th} iteration of the merge step. A potential performance improvement may come from the possibility of performing vector-matrix multiplications instead of matrix-matrix multiplications.

Although most performance assessments in Turin (1998) focused on parallel versions of the complete Baum-Welch algorithm, it was also noted that calculating $\mathbf{P}(\bullet)$ for blocks requires $\mathcal{O}(TS^3)$ operations, while performing the sequential merge requires $\mathcal{O}(TS^2)$ operations.

The formulation of the recursions suggest the possibility of online applications of the Baum-Welch algorithm (and the backward recursions and the forward-backward algorithm).

Intuition and limitations

If we generalise Turin's formulas for inhomogeneous HMMs, it is clear that every $\mathbf{P}(y_t)$ encapsulates all the information introduced at every t (including the potentially t-dependent transitions $\mathbf{A_t}$ and emissions $\mathbf{E_t^d}$) but contain no information from preceding or succeeding time points.

From the definition of $\mathbf{P}(y_t)$ it is clear that calculations at different t are indeed independent and, hence, may be performed in parallel, with the actual alpha results acquired by a merge

step (e.g. sequentially performing the vector-matrix multiplications from $t=1$ to $t=T$ for the forward algorithm).

By calculating the $\mathbf{P}(y_t)$ and performing merge afterwards, calculations are separated into independent and dependent parts - the independent calculations (calculating $\mathbf{P}(y_t)$) may be performed completely in parallel, while calculations that introduce dependencies are effectively postponed to the merge step, where the cost of restoring dependencies is the necessity to perform sequential calculation (in sequential merge) or extra operations (more than S-times more operations with matrix-matrix multiplications in each parallel merge).

One immediate benefit of using Turin's methods is the possibility to find $\mathbf{P}(y_{t_1:t_2})$ for long blocks of identical observations using fast matrix exponentiation. However, not counting the accelerations from fast matrix exponentiation, the parallelisation approaches presented are limited in terms of parallel acceleration. Calculating $\mathbf{P}(y_t)$ takes $\mathcal{O}(S^2)$ operations at each $t \in \{1, ..., T\}$ and each merge step also requires $\mathcal{O}(S^2)$ calculations. Consequently, even if a $C \geq T$ parallelism limit is assumed, run time will be $\mathcal{O}(S^2+TS^2)$, and this parallelisation approach may only yield some $\mathcal{O}(1)$ acceleration in its basic form when compared to the sequential version ($\mathcal{O}(TS^2)$). In fact, Turin noted that the main benefit of the forward-only and backward-only algorithms are in space requirement reductions and not in significant parallel accelerations.

Turin argued that even with the availability of two processors ($C=2$), the sequential algorithm would waste resources and a three-node tree-based reduction would perform better. While that may be the case for $C=2$, where a single-block forward-only and a single-block backward-only algorithm may be used, this approach can not be scaled for higher numbers of blocks ($B>2$ when $C, T>2$) without re-introducing either sequential or tree-reduction-based merge.

2.2 Tree-reduction, improved sequential merge, matrix Viterbi and checkpointing

Tree reduction re-discovered

Nielsen and Sand (2011) explored block-parallelisation based on the associativity of matrix multiplications and appear to have re-discovered the tree-reduction based merge step suggesting that the multiplications in $\prod \mathbf{P}(y_{a:b})$ may be grouped into a binary tree as

$$\prod_{t=a}^{b} \mathrm{P}(y_t) = \Big(\Big(\big(\mathbf{P}(y_1)\,\mathbf{P}(y_2)\big)\,\big(\mathbf{P}(y_3)\,\mathbf{P}(y_4)\big) \Big) \;\cdots\; \big(\mathbf{P}(y_{2k-1})\,\mathbf{P}(y_{2k})\big) \;\cdots \Big),$$

which is essentially identical to (IV.6). While this algorithm may achieve an $\mathcal{O}(S^3 \log T)$ theoretical execution time, Nielsen and Sand acknowledged that it introduces an $\mathcal{O}(S)$ factor increase in computation (requiring $\mathcal{O}(S^3T)$ operations because of the matrix-matrix multiplications). It was also noted that for a T-fold parallelism it would be necessary to have a 'processor' for every t (meaning $C \geq T$), which is not expected to be the case in practice. Consequently, the tree-reduction based parallel merge step is disregarded for practical use.

Improved sequential merge for multi-core systems

Nielsen and Sand also observed the $\mathcal{O}(S)$ factor of computational cost difference between matrix-matrix and vector-matrix multiplications and suggested a slightly optimised sequential merge step for the forward algorithm (which was called as the 'parredForward' algorithm). In this version the parallel computations are performed on blocks ($B, L_b > 1$) using matrix-matrix multiplications only on blocks $b > 1$. Merge is performed with matrix-vector multiplications from the first block $b = 1$ to $b = B$, but with the size of the first block being S-times longer than the size of later blocks. The advantage of this approach is that the thread or processor that performs a normal forward algorithm on the first block[5] is not idle while the more lengthy computations on succeeding blocks are performed. This slightly optimised algorithm may perform with $\mathcal{O}(S^3T/(S+C-1))$ asymptotic run time, achieving an overall $X = 1+(C-1)/S$ acceleration. This result is relevant mostly when C is limited.

Matrix Viterbi

The matrix block-parallelisation of the Viterbi algorithm (called 'parredViterbi' in Nielsen and Sand (2011)) builds on the associativity of the 'matrix-max' operation $A \otimes B$, where

$$\big(\mathbf{A} \otimes \mathbf{B}\big)_{ij} = \max_k \{A_{ik} B_{kj}\}. \tag{IV.7}$$

The Viterbi recursion is formulated as

$$\boldsymbol{\delta}_t \;=\; \{\delta s_t\} \;=\; \boldsymbol{\delta_{t-1}} \otimes \mathbf{P}(y_t) = \boldsymbol{\pi}\, \mathbf{P}(y_1) \otimes \mathbf{P}(y_2) \otimes \;\cdots\; \otimes \mathbf{P}(y_t) \otimes \mathbf{P}(y_t) \tag{IV.8}$$

and for a block $\{k, ..., l\}$, the matrix

$$\mathbf{D}_{k:l} \;\triangleq\; \mathbf{P}(y_k) \otimes \mathbf{P}(y_{k+1}) \otimes \;\cdots\; \otimes \mathbf{P}(y_{l-1}) \otimes \mathbf{P}(y_l) \tag{IV.9}$$

is defined.

[5] In Nielsen and Sand (2011), a maximum of one processor or thread is assumed for each block.

The matrix Viterbi algorithm is based on the observation that

$$\boldsymbol{\delta}_t \;=\; \boldsymbol{\pi}\,\mathbf{D}_{1:t} \;=\; \boldsymbol{\pi}\,\mathbf{D}_{1:k} \otimes \mathbf{D}_{k+1:t}, \tag{IV.10}$$

and is performed by calculating $\mathbf{D}_{\tau_b:\tau'_b}$ for consecutive blocks (and $\delta(s_{\tau'_1})$ for the end of the first block), from which $\boldsymbol{\delta}_{\tau'_b}$ is found for each block-end consecutively. It is possible to achieve some acceleration and space requirement reductions by calculating results only for block-ends ($t=\tau'_b$) in the sequential merge step and then re-running normal Viterbi algorithms in parallel in sub-blocks (now with the knowledge of the $\delta(s_{\tau_b})$) to find the Viterbi path (s^*_t) inside blocks.

With the first block chosen to be S-times longer than other blocks, the matrix Viterbi is also expected to have an $\mathcal{O}(S^3T/(S+C-1))$ run time.

Intuition and limitations

Although most of the ideas presented in Nielsen and Sand (2011) are essentially identical to those in Turin (1998), there is a slightly more detailed explanation of the tree-based merge step, the importance of larger blocks ($L_b > 1$) is realised and two possible optimisations are introduced. While setting $L_1 = SL_b$, for $b > 1$ may bring only minor performance improvements, the acceleration of sequential merge achieved for the matrix Viterbi algorithm with the use of checkpointing (see Grice et al. (1997) and Tarnas and Hughey (1998) for original works on checkpointing for sequential HMM algorithms) may have potential in large-scale HMM parallelisation approaches for the forward and backward recursions as well.

3 A probabilistic-algorithmic view of HMM block-parallelism

This section presents a probabilistic-algorithmic[6] view of HMM block-parallelism. The probabilistic view is presented with the belief that it is often easier and more natural to start method development with a probabilistic understanding. It is particularly important when development involves modified HMMs or modified HMM algorithms (e.g. variable-order HMMs, lazy Viterbi, etc.). It will be shown in section 4 that all presented algorithms can be expressed in matrix form (similarly to the approaches presented in prior works - see Section 2), which allows for simpler practical implementations as well as the use of available parallel linear algebra libraries (such as cuBLAS for CUDA).

For simplicity, $L=T$ is assumed throughout this section.

[6]This framework may be seen as algorithmic as it is somewhat based on following the information flow.

3.1 HMM block-parallelism with simple sequential merge

It is clear that running HMM algorithms independently on separate observation blocks is trivial. The task in block-parallelisation is to perform independent calculations on different blocks in a way such that it is later possible to assemble correct results by recovering the sequential dependencies between consecutive blocks.

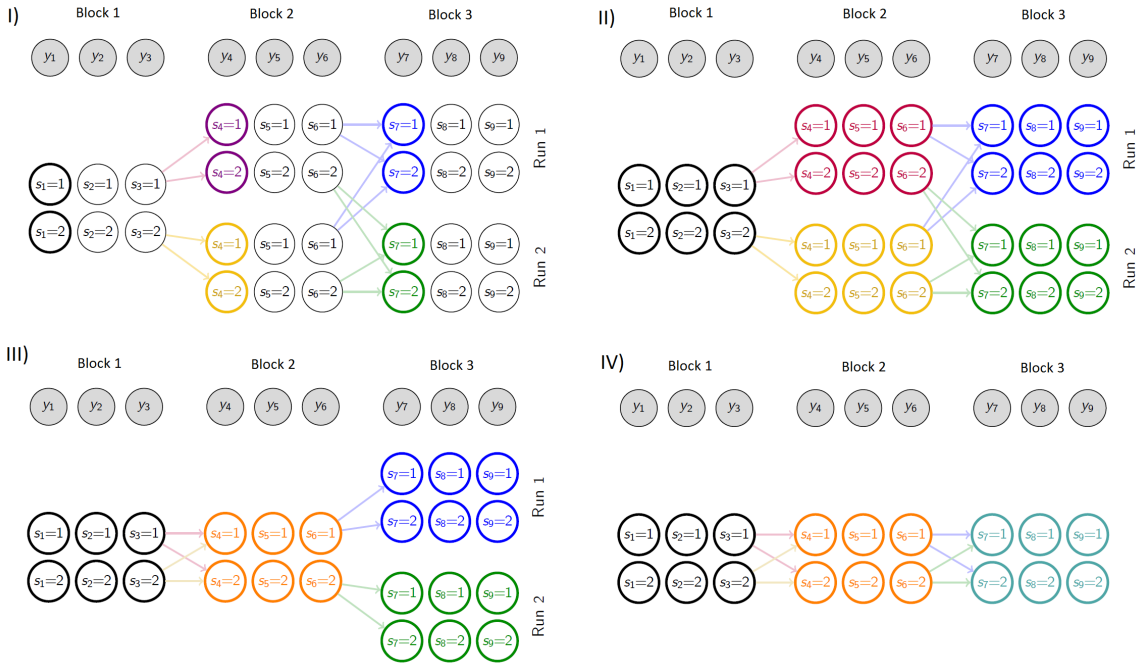

Figure IV.4: Parallelising HMM algorithms over blocks of the observation length with sequential merge. The number of hidden states is S=2 and the number of observations is K=1, with T=9 pieces of observation that are partitioned into B=3 blocks of length T_b=3. The sub-figures demonstrate various stages of the algorithm. (**I**) presents the initialisation of sub-runs (called 'Run 1' and 'Run 2') and the normal run on the first block. (**II**) shows the stage when the parallel runs are completed and all block-wise results are available. (**III**) presents the stage when blocks 1 and 2 are merged and (**IV**) shows the final stage when the correct results are recovered for all blocks.

The foundation of BP is the Markov property of HMMs: calculations in a block are dependent only on the state at the end of the previous block or the beginning of the next block. If the forward algorithm is taken as an example for block b (ranging $t \in \{\tau_b, ..., \tau'_b\}$), the recursion values (the $\alpha(s_t)$ messages) at $t \geq \tau_b$ are independent of those at $t < \tau'_{b-1}$ given the results at τ'_{b-1}. This means that the computations on block b only require knowledge of the results $\alpha(s_{\tau'_{b-1}})$ at the last t of the previous block. Consequently, the independence can be exploited by running multiple runs (a bundle $\mathcal{R}_b$ of $|\mathcal{R}_b|$=S sub-runs) of HMM algorithms on each block, with each sub-run $r \in \mathcal{R}_b$ conditioned on the HMM being in state $s_{\tau'_{b-1}} = r$ at the end of the previous block. In case of the backward pass in the forward-backward algorithm, sub-runs may be conditioned on states at the beginning of succeeding blocks ($s_{\tau_{b+1}}$=r).

After acquiring results from all S sub-runs, the correct results can be recovered by letting each r sub-run contribute to the reconciled final results according to the messages at block boundaries (e.g. in the case of the forward algorithm, according to the probability that the HMM was in state r at the end of the previous block). With only a standard sequential recursion performed on the first (or last) block, the sub-run results of block 2 (or B–1) can be merged with the finalised results of block 1 (or B). Next, the joint results from these blocks can be merged with the next block and so on, sequentially (see Fig. IV.4 and Algorithm 9).

Algorithm 9 Skeleton pseudocode for block-parallel HMM algorithms following the probabilistic framework with sequential merge. The example corresponds to the forward algorithm.

1: **parallel for all** blocks $b > 1$ **do**
2: **parallel for all** sub-run $r \in \{1, ..., S\}$ **do**
3: Run the forward pass from τ_b to τ'_b conditioning on $s_{\tau'_{b-1}}{=}r$ to obtain results for sub-run r;
4: **end parallel**
5: **end parallel**
6: **for all** blocks $b{>}1$ **do**
7: Merge the finalised results from $b{-}1$ and the conditional results from the sub-runs of b;
8: **end for**

In order to perform calculations over B blocks in the parallel recursion step, the minimum effective parallel processing requirement is $C \geq B$, where B is essentially a parallelisation parameter that can be chosen (and optimised for a given pair of S and T).

3.1.1 The forward algorithm

This section considers the forward algorithm in its "stand alone" version, i.e. when the objective is to compute the likelihood $\mathrm{P}(y_{1:T}|\boldsymbol{\lambda})$ or the hidden state probabilities at the last time step $\mathrm{P}(s_T|y_{1:T}, \boldsymbol{\lambda})$.

The standard sequential forward algorithm

The standard sequential forward algorithm calculates $\alpha(s_t){=}\mathrm{P}(y_{1:t}, s_t)$ with a single forward pass through each position t according to the recursion

$$\alpha(s_t) = \mathrm{P}(y_t|s_t) \sum_{s_{t-1} \in S} \mathrm{P}(s_t|s_{t-1})\, \alpha(s_{t-1}) \qquad \text{for } t > 1 \tag{IV.11}$$

$$\alpha(s_1) = \mathrm{P}(y_1|s_1)\, \pi(s_1). \tag{IV.12}$$

The recursion has $\mathcal{O}(S^2T)$ computation and runtime costs and an $\mathcal{O}(S)$ space requirement.

The forward algorithm parallelised over observation blocks

In the block-parallel version, the forward algorithm is first run on each block b independently (and in parallel if C permits). For each block $b>1$, a bundle $\mathcal{R}_b$ of $|\mathcal{R}_b| = S$ sub-runs are launched with each $r \in \mathcal{R}_b$ sub-run assuming $s_{\tau'_{b-1}} = r$ to be fixed, thus calculating results conditioned on the state at the end of the previous block. Correspondingly, the conditional forward messages for the current block, with $t \in \{\tau_b, ..., \tau'_b\}$ are defined to be

$$\alpha_r^{(b)}(\, s_t \,|\, s_{\tau'_{b-1}} = r) \;\triangleq\; \mathrm{P}(\, y_{\tau_b:t},\, s_t \,|\, s_{\tau'_{b-1}} = r). \tag{IV.13}$$

Following from this definition of conditional messages, the conditional alpha messages in each r run in block $b>1$ are calculated as

$$\alpha_r^{(b)}(\, s_{\tau_b} \,|\, s_{\tau'_{b-1}} = r) \;=\; \mathrm{P}(\, y_{\tau_b} \,|\, s_{\tau_b})\, \mathrm{P}(\, s_{\tau_b} \,|\, s_{\tau'_{b-1}} = r) \tag{IV.14}$$

for sub-run initialisations ($t = \tau_b$) and

$$\alpha_r^{(b)}(s_t \,|\, s_{\tau'_{b-1}} = r) \;=\; \mathrm{P}(\, y_t \,|\, s_t) \sum_{s_{t-1}} \mathrm{P}(\, s_t \,|\, s_{t-1})\;\; \alpha_r^{(b)}(\, s_{t-1} \,|\, s_{\tau'_{b-1}} = r) \tag{IV.15}$$

for the rest of each block, $t \in \{\tau_b{+}1, .., \tau'_b\}$. Equations (IV.14) and (IV.15) can be seen as conditional versions of equations (IV.11) and (IV.12).

The alpha messages $\alpha(s_{\tau'_1})$ of the first block can be calculated with a single ordinary forward pass on the first block. When the independent sub-run computations are finished on blocks $b{=}2$, and the results from the first block are also available, it becomes possible to perform the first merge step, but it may also be postponed until results are ready for all blocks $b>1$. The final alpha messages $\alpha(s_{\tau'_b})$ at the end of block $b{>}1$ can be calculated as

$$\begin{aligned}
\alpha(s_{\tau'_b}) \;&=\; \mathrm{P}(\, y_{1:\tau'_b},\; s_{\tau'_b}) \;=\; \mathrm{P}(\, y_{1:\tau'_{b-1}},\; y_{\tau_b:\tau'_b},\; s_{\tau'_b}) && \text{(IV.16a)}\\
&=\; \sum_{s_{\tau'_{b-1}}} \mathrm{P}(\, y_{\tau_b:\tau'_b},\; s_{\tau'_b},\; y_{1:\tau'_{b-1}},\; s_{\tau'_{b-1}}) && \text{(IV.16b)}\\
&=\; \sum_{s_{\tau'_{b-1}}} \mathrm{P}(\, y_{\tau_b:\tau'_b},\; s_{\tau'_b} \,|\, y_{1:\tau'_{b-1}},\; s_{\tau'_{b-1}})\;\; \mathrm{P}(\, y_{1:\tau'_{b-1}},\; s_{\tau'_{b-1}}) && \text{(IV.16c)}\\
&=\; \sum_{s_{\tau'_{b-1}}} \mathrm{P}(\, y_{\tau_b:\tau'_b},\; s_{\tau'_b} \,|\, s_{\tau'_{b-1}})\;\; \mathrm{P}(\, y_{1:\tau'_{b-1}},\; s_{\tau'_{b-1}}) && \text{(IV.16d)}\\
&=\; \sum_{r} \alpha_r^{(b)}(s_{\tau'_b} \mid s_{\tau'_{b-1}} = r)\;\; \alpha(s_{\tau'_{b-1}} = r) && \text{(IV.16e)}
\end{aligned}$$

iteratively from $b{=}2$ to $b{=}\mathrm{B}$. The $\alpha(s_{\tau'_b})$ results may be seen as a weighted sum over corre-

sponding sub-run results, where the weights are given as the probability that the sub-runs' assumptions ($\mathrm{P}(s_{\tau'_{b-1}}=r,\ y_{1:\tau'_{b-1}}) = \alpha(s_{\tau'_{b-1}}=r)$) are correct.

Algorithm 10 Complete pseudocode for the block-parallel forward algorithm with sequential merge. In this version only the forward pass is parallelised and only over the observation blocks.

1: Run a single standard sequential forward pass on block 1 with π;
2: **parallel for all** blocks $b \in \{2, .., B\}$ **do** ▷ The parallel runs
3: **for all** sub-runs $r \in \mathcal{R}_b$ **do**
4: Run a forward pass assuming $s_{\tau'_{b-1}}=r$ to get $\alpha_r^{(b)}(s_{\tau'_b}|s_{\tau'_{b-1}}=r), \quad t \in \boldsymbol{t}_b$; ▷ eqs. (IV.14), (IV.15)
5: **end for**
6: **end parallel**
7: **for all** blocks $b \in \{2, .., B\}$ **do** ▷ The merge step
8: **for all** $s_{\tau'_b} \in \mathcal{S}$ **do**
9: Obtain $\alpha(s_{\tau'_b})$ from $\alpha(s_{\tau'_{b-1}})$ and the sub-run results $\alpha_r^{(b)}(s_{\tau'_b}|s_{\tau'_{b-1}})$ of block b; ▷ eq. (IV.16)
10: **end for**
11: **end for**

With every step and calculation defined, it is now possible to assemble a complete pseudocode for the block-parallel forward algorithm (see Algorithm 10). Line 4 of the algorithm corresponds to individual sub-runs with $\mathcal{O}(L_b S^2)$ operations each on an L-long section, and lines 8-10 correspond to individual merge steps, each requiring $\mathcal{O}(S^2)$ operations ($\mathcal{O}(BS^2)$ in total). The overall run time is $\mathcal{O}(L_b S^3 + BS^2)$ for a section in the most basic case (i.e. without further parallelism), provided that $C \geq B$.

When the block-parallel forward algorithm is used to calculate results only for $t = T$, the space requirement is $\mathcal{O}(BS^2)$, because $\alpha_r^{(b)}(s_t|s_{\tau'_{b-1}})$ is saved for all states in all sub-runs in each block during the block-parallel recursion.

3.1.2 The forward-backward algorithm

The standard sequential forward-backward algorithm

The forward-backward algorithm calculates the marginal state probabilities at each t as

$$\gamma(s_t) \ = \ \mathrm{P}(\ s_t \mid y_{1:T}, \boldsymbol{\lambda}) \ = \ \frac{\alpha(s_t)\ \beta(s_t)}{\mathrm{P}(\ y_{1:T} \mid \boldsymbol{\lambda})} \ = \ \frac{\alpha(s_t)\ \beta(s_t)}{\sum_{s_t=1}^{S} \alpha(s_t)\ \beta(s_t)} \tag{IV.17}$$

where the $\alpha(s_t)$ are calculated with a forward pass as in the forward algorithm (eqs. (IV.11) and (IV.12)).

The backward messages $\beta(s_t) = \mathrm{P}(y_{t+1:T}, s_t)$ are calculated with a backward recursion (the backward pass) as

$$\beta(s_t) = \sum_{s_{t+1}} \Big(\mathrm{P}(\, y_{t+1} \,|\, s_{t+1}\,)\;\; \mathrm{P}(\, s_{t+1} \,|\, s_t\,)\, \beta(s_{t+1}) \Big), \tag{IV.18}$$

$$\beta(s_T) = 1. \tag{IV.19}$$

The standard forward-backward algorithm requires $\mathcal{O}(S^2T)$ computation and run time and $\mathcal{O}(ST)$ space.

The block-parallel forward-backward algorithm

Since the forward-backward algorithm can be executed by first running a forward pass and then running a backward pass, the two can be parallelised separately. The forward pass is parallelised as in the case of the forward algorithm, but with $\alpha(s_t|s_{\tau'_{b-1}})$ stored for $\forall t \in \{1, ..., T_b\}$ in each block (not just for the end of each block τ'_b). The backward pass is parallelised using the 'conditional backward messages'

$$\beta_r^{(b)}(\, s_t \,|\, s_{\tau_{b+1}}{=}r) \;\triangleq\; \mathrm{P}(\, y_{t+1:\tau'_b},\, s_t \,|\, s_{\tau_{b+1}}{=}r) \tag{IV.20}$$

and similarly to the block-parallel forward pass, the backward pass starts with performing a standard backward pass on block B and conditional sub-runs on each block $b{<}B$, calculating

$$\beta_r^{(b)}(s_t \,|\, s_{\tau_{b+1}}{=}r) \;\;=\;\; \mathrm{P}(\, y_t \,|\, s_t)\, \sum_{s_{t+1}} \mathrm{P}(\, s_{t+1} \,|\, s_t)\;\; \beta_r^{(b)}(\, s_{t+1} \,|\, s_{\tau_{b+1}}{=}r) \tag{IV.21}$$

for $t \in \{\tau_b, .., \tau'_b{-}1\}$, and

$$\beta_r^{(b)}(\, s_{\tau'_b} \,|\, s_{\tau_{b+1}}{=}r) \;\;=\;\; \mathrm{P}(\, y_{\tau'_b} \,|\, s_{\tau'_b})\, \mathrm{P}(\, s_{\tau_{b+1}} \,|\, s_{\tau_{b+1}}{=}r) \tag{IV.22}$$

for sub-run initialisations ($t{=}\tau'_b$).

In the merge step, the $\beta(s_t)$ in block b are then reconstructed using the sub-run results $\beta_r^{(b)}(\, s_t \,|\, s_{\tau_{b+1}}{=}r)$ and the results $\beta(s_{\tau_{b+1}})$ from block $b{+}1$ as

$$\begin{aligned}
\beta(s_t) &= \mathrm{P}(\, y_{t+1:T},\, s_t) \;=\; \mathrm{P}(\, y_{t+1:\tau_b'},\, y_{\tau_{b+1}:T}\,,\, s_t) && \text{(IV.23a)}\\
&= \sum_{s_{\tau_{b+1}}} \mathrm{P}(\, y_{t+1:\tau_b'}\,,\, s_t\,,\, y_{\tau_{b+1}:T},\, s_{\tau_{b+1}}) && \text{(IV.23b)}\\
&= \sum_{s_{\tau_{b+1}}} \mathrm{P}(\, y_{t+1:\tau_b'}\,,\, s_t \mid y_{\tau_{b+1}:T},\, s_{\tau_{b+1}})\;\; \mathrm{P}(y_{\tau_{b+1}:T},\, s_{\tau_{b+1}}) && \text{(IV.23c)}\\
&= \sum_{s_{\tau_{b+1}}} \mathrm{P}(\, y_{t+1:\tau_b'}\,,\, s_t \mid s_{\tau_{b+1}})\;\; \mathrm{P}(y_{\tau_{b+1}:T},\, s_{\tau_{b+1}}) && \text{(IV.23d)}\\
&= \sum_{r} \beta_r^{(b)}(s_t \mid s_{\tau_{b+1}}{=}\, r)\;\; \beta(s_{\tau_{b+1}}{=}\, r). && \text{(IV.23e)}
\end{aligned}$$

The complete pseudocode of the block-parallel backward pass is presented as Algorithm 11. When used in the forward-backward algorithm, the forward pass (Algorithm 10) needs to be modified to also calculate α_{s_t} for each $t \in \{\tau_b, ..., \tau_b'\}$ in each block. This way the parallel sub-runs execute in $\mathcal{O}(L_b S^3)$ time and the merge steps cost $\mathcal{O}(L_b S^2)$ for each block. Overall execution time is $\mathcal{O}(L_b S^3{+}BL_b S^2) = \mathcal{O}(L_b S^3{+}LS^2)$ for a single section, assuming no further parallelisation and assuming $C \geq B$. The computation cost is $\mathcal{O}(LS^3 + BL_b S^2) = \mathcal{O}(LS^3)$.

Algorithm 11 Complete pseudocode for the block-parallel backward algorithm with sequential merge. Only the backward pass is parallelised and only over the observation blocks.

```
Run a single standard backward pass on block B;
parallel for all blocks b ∈ {1, .., B−1} do                          ▷ The parallel runs
    for all sub-runs r ∈ R_b do
        Run a backward pass assuming s_{τ_{b+1}}=r to find all β_r^(b)(s_t|s_{τ_{b+1}}),  t∈t_b ;   ▷ eqs. (IV.21), (IV.22)
    end for
end parallel
for all blocks b ∈ {1, .., B − 1} do                                 ▷ The merge step
    for all t ∈ {τ_b, ..., τ_b'} do
        for all s_t ∈ S do
            Obtain β(s_t) from β(s_{τ_{b+1}}) and the sub-run results β_r^(b)(s_t|s_{τ_{b+1}}) of block b;   ▷ eq. (IV.23)
        end for
    end for
end for
```

In the basic version of the forward-backward algorithm it is necessary to save all $\alpha(s_t)$ (or all $\beta(s_t)$) for each t in each block, each sub-run and every possible state. Hence the space requirement is $\mathcal{O}(TS^2)$.

3.1.3 The Viterbi algorithm

The standard sequential Viterbi algorithm

The Viterbi algorithm calculates the most probable state sequence (the Viterbi path) for a given observation in two stages. First, the algorithm calculates

$$\delta(s_t) = \max_{s_{1:t-1}} \mathrm{P}(s_{1:t}, y_{1:t}|\boldsymbol{\lambda}) \tag{IV.24}$$

and saves the corresponding backtrace pointers

$$\arg\max_{s_{t-1}} \left\{ \mathrm{P}(\, s_t \,|\, s_{t-1}\,)\; \delta(s_{t-1}) \right\} \tag{IV.25}$$

in a backtrace matrix for each s_t. In the second stage, the Viterbi path is assembled by following the backtrace from $\arg\max_{s_T}\{\delta(s_T)\}$.

The recursion in the first stage of the Viterbi algorithm is performed as

$$\delta(s_t) = \mathrm{P}(\, y_t \,|\, s_t\,) \;\; \max_{s_{t-1}} \left\{ \mathrm{P}(\, s_t \,|\, s_{t-1}\,)\; \delta(s_{t-1}) \right\} \tag{IV.26}$$

$$\delta(s_1) = \pi(s_1)\; \mathrm{P}(\, y_1 \,|\, s_1\,). \tag{IV.27}$$

The recursion in the standard sequential Viterbi algorithm requires $\mathcal{O}(S^2T)$ computation and run time and $\mathcal{O}(ST)$ space. The backtrace takes $\mathcal{O}(T)$ iterations.

The block-parallel Viterbi algorithm

The backtrace stage of the Viterbi algorithm is inherently sequential and relatively fast, hence it is best to focus parallelisation on the recursion. The recursion is parallelised as in the case of the forward algorithm, but now using 'conditional Viterbi messages' defined as

$$\delta_r^{(b)}(\, s_t \,|\, s_{\tau'_{b-1}}{=}r) \;\triangleq\; \max_{s_{\tau_b:t-1}} \; \mathrm{P}(\, s_{\tau_b:t},\, y_{\tau_b:t} \,|\, s_{\tau'_{b-1}}{=}r). \tag{IV.28}$$

The block-parallel recursion starts with performing conditional sub-runs on each block $b{>}1$

$$\delta_r^{(b)}(s_t \,|\, s_{\tau'_{b-1}}{=}r) \;\; = \;\; \mathrm{P}(\, y_t \,|\, s_t) \;\; \max_{s_{t-1}} \left\{ \mathrm{P}(\, s_t \,|\, s_{t-1}) \;\; \delta_r^{(b)}(\, s_{t-1} \,|\, s_{\tau'_{b-1}}{=}r) \right\} \tag{IV.29}$$

for $t \in \{\tau_b{+}1, .., \tau'_b\}$, and

$$\delta_r^{(b)}(\, s_{\tau_b} \,|\, s_{\tau'_{b-1}}{=}r) \;\; = \;\; \mathrm{P}(\, y_{\tau_b} \,|\, s_{\tau_b}) \;\; \mathrm{P}(\, s_{\tau_b} \,|\, s_{\tau'_{b-1}}{=}r) \tag{IV.30}$$

for sub-run initialisations ($t{=}\tau'_b$).

In the merge step of the block-parallel Viterbi algorithm, the $\delta(s_t)$ for block b and $t \in \{\tau_b, ..., \tau'_b\}$ are constructed using the sub-run results $\delta_r^{(b)}(\, s_t \,|\, s_{\tau'_{b-1}}{=}r)$ from block b and the final results $\delta(s_{\tau'_{b-1}})$ from block $b{-}1$.

For block boundaries ($t{=}\tau_b$ and $t{-}1 = \tau'_{b-1}$),

$$\begin{aligned}
\delta(s_{\tau_b}) \;\; &= \;\; \mathrm{P}(y_{\tau_b}|s_{\tau_b}) \max_{s_{\tau'_{b-1}}} \left\{ \;\; \mathrm{P}(s_{\tau_b}|s_{\tau'_{b-1}}) \;\; \delta(s_{\tau'_{b-1}}) \;\; \right\} && \text{(IV.31a)} \\
&= \;\; \max_{s_{\tau'_{b-1}}} \left\{ \;\; \mathrm{P}(y_{\tau_b}|s_{\tau_b}) \, \mathrm{P}(s_{\tau_b}|s_{\tau'_{b-1}}) \;\; \delta(s_{\tau'_{b-1}}) \;\; \right\} && \text{(IV.31b)} \\
&= \;\; \max_{s_{\tau'_{b-1}}} \left\{ \;\; \delta_r^{(b)}\,(s_{\tau_b}|s_{\tau'_{b-1}}) \;\; \delta(s_{\tau'_{b-1}}) \;\; \right\} && \text{(IV.31c)} \\
&= \;\; \max_{r} \left\{ \;\; \delta_r^{(b)}\,(s_{\tau_b}|s_{\tau'_{b-1}}{=}r) \;\; \delta(s_{\tau'_{b-1}}{=}r) \;\; \right\} && \text{(IV.31d)}
\end{aligned}$$

and for $t \in \{\tau_b{+}1, ..., \tau'_b\}$,

$$\delta(s_t) \;\; = \;\; \max_{r} \left\{ \;\; \delta_r^{(b)}(\; s_t \,|s_{\tau'_{b-1}}{=}r\,) \;\; \delta(s_{\tau'_{b-1}}{=}r) \;\; \right\}. \tag{IV.32}$$

The backtrace pointers

$$\operatorname*{arg\,max}_{s_{t-1}} \;\; \left\{ \;\; \mathrm{P}(s_t|s_{t-1}) \;\; \delta(s_{t-1}) \;\; \right\} \tag{IV.33}$$

are found and stored as normal. They can be calculated in the merge step (if the merge is not parallelised over $t \in \{\tau_b, ..., \tau'_b\}$) or in a separate sweep once all $\delta(s_t)$ are available for the block (which is useful for parallelisation). It is also possible to follow a checkpointing strategy, where the $\delta(s_t)$ are calculated in a post-processing step (as in Nielsen and Sand (2011)).

The complete pseudocode is presented as Algorithm 12 – the parallel sub-runs execute in $\mathcal{O}(L_b S^3)$ time and the merge steps require $\mathcal{O}(L_b S^2)$ operations in each block in a section. Overall execution time is $\mathcal{O}(L_b S^3{+}LS^2)$ without further parallelisations and the computation cost is $\mathcal{O}(BL_b S^3 + BL_b S^2) = \mathcal{O}(LS^3)$.

The space requirement of the block-parallel Viterbi algorithm is $\mathcal{O}(BL_b S^2) = \mathcal{O}(LS^2)$, since the $\delta_r^{(b)}$ have to be stored for all states s_t in all sub-runs for each piece of observation in each

block.

Algorithm 12 Complete pseudocode for the block-parallel Viterbi algorithm with sequential merge. Only the recursion is parallelised and only over the observation blocks.

Run a standard Viterbi algorithm once on block b=1;
parallel for all blocks $b \in \{2, .., B\}$ **do** ▷ The parallel runs
  for all sub-runs $r \in \mathcal{R}_b$ **do**
    Run the Viterbi alg. assuming $s_{\tau'_{b-1}}=r$ to get $\delta_r^{(b)}(s_t|s_{\tau'_{b-1}}=r), \quad t \in \boldsymbol{t}_b$; ▷ eqs. (IV.29), (IV.30)
  end for
end parallel
for all blocks $b \in \{2, .., B\}$ **do** ▷ The merge step
  Calculate $\delta(s_{\tau_b})$ and the corresponding backtrace pointer and save; ▷ eqs. (IV.31), (IV.33)
  for all $t \in \{\tau_b+1, ..., \tau'_b\}$ **do**
    for all $s_t \in \mathcal{S}$ **do**
      Calculate $\delta(s_t)$, find the corresponding backtrace pointer and save; ▷ eqs. (IV.32), (IV.33)
    end for
  end for
end for
Find the Viterbi path backward from $\arg\max_{s_T} \delta(s_T)$ using the backtrace pointers as normal.

.

3.2 HMM block-parallelism with parallel merge

All derivations in Section 3.1 assume that the merge step is performed sequentially over blocks and sub-run results from each block b are joined with the finalised results corresponding to all preceding blocks $1, ..., b$–1. The advantages of the sequential merge strategy are simplicity and that merge steps cost only $\mathcal{O}(S^2)$ operations each. The disadvantage is that the order of merge steps is fixed due to their sequential dependency and they can not be run in parallel.

In order to be able to perform merge operations in any order (independently and potentially in parallel), the merge step has to be performed using results of sub-runs from one block and results of sub-runs from a neighbouring block. The results of these merge steps will be sub-run results corresponding to the joint (concatenated) block (see Figure IV.5 and note that different results are retained for sub-runs, unlike with sequential merge). The corresponding merge step for blocks $b-1$ and b in the case of the forward recursion is

$$
\begin{aligned}
\alpha_r^{(b-1,b)}(s_t \mid s_{\tau'_{b-2}}) &= \mathrm{P}(\, y_{\tau_{b-1}:t},\ s_t \mid s_{\tau'_{b-2}}) \;=\; \mathrm{P}(\, y_{\tau_{b-1}:\tau'_{b-1}},\ y_{\tau_b:t},\ s_t \mid s_{\tau'_{b-2}}) \\
&= \sum_{s_{\tau'_{b-1}}} \mathrm{P}(\, y_{\tau_{b-1}:\tau'_{b-1}},\ s_{\tau'_{b-1}},\ y_{\tau_b:t},\ s_t \mid s_{\tau'_{b-2}}) \\
&= \sum_{s_{\tau'_{b-1}}} \Big\{ \mathrm{P}(\, y_{\tau_b:t},\ s_t \mid y_{\tau_{b-1}:\tau'_{b-1}},\ s_{\tau'_{b-1}},\ s_{\tau'_{b-2}})\ \mathrm{P}(\, y_{\tau_{b-1}:\tau'_{b-1}},\ s_{\tau'_{b-1}} \mid s_{\tau'_{b-2}}) \Big\} \\
&= \sum_{s_{\tau'_{b-1}}} \Big\{ \mathrm{P}(\, y_{\tau_b:t},\ s_t \mid s_{\tau'_{b-1}})\ \mathrm{P}(\, y_{\tau_{b-1}:\tau'_{b-1}},\ s_{\tau'_{b-1}} \mid s_{\tau'_{b-2}}) \Big\} \\
&= \sum_{q} \Big\{ \alpha_q^b\,(s_t \mid s_{\tau'_{b-1}} = q)\ \ \alpha_r^{b-1}\,(s_{\tau'_{b-1}} = q \mid s_{\tau'_{b-2}} = r) \Big\}, \qquad \text{(IV.34)}
\end{aligned}
$$

where r is the index of sub-runs on block $b-1$ (and refers to the condition on $s_{\tau'_{b-2}}$), q is the index of sub-runs on block b (and refers to the condition on $s_{\tau'_{b-1}}$) and $\alpha_r^{(b-1,b)}(s_t \mid s_{\tau'_{b-2}})$ is a matrix of $S \times S$ values.

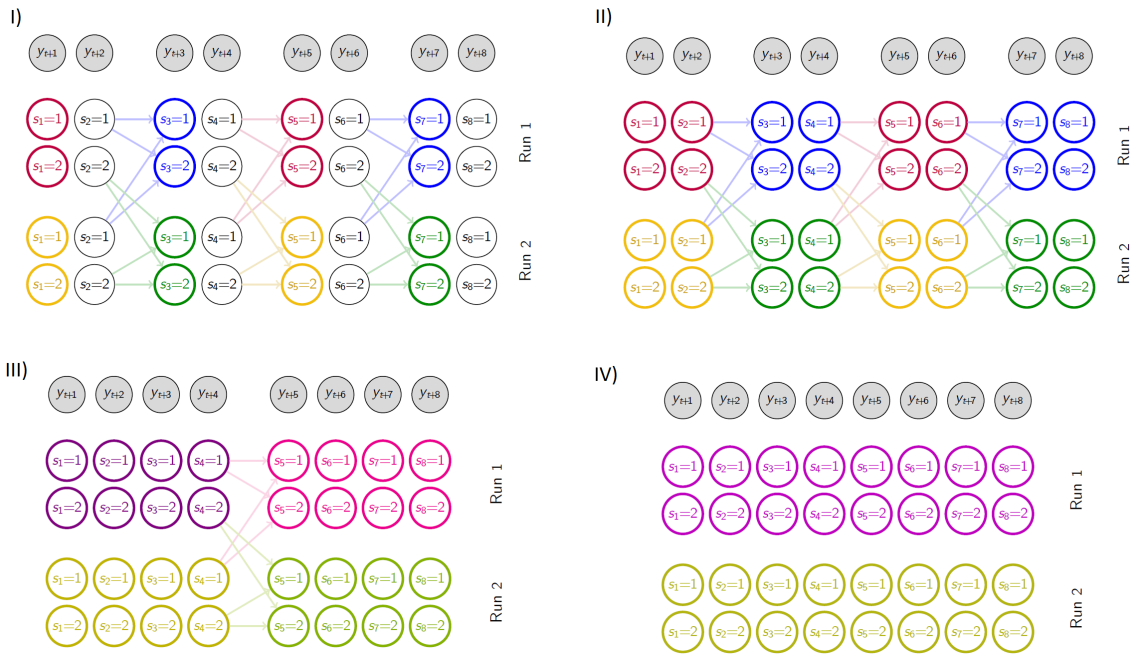

Figure IV.5: Parallel merge for block-parallel HMM algorithms. The number of hidden states is S=2 and the number of observations is K=1, with the section $t+1,..,t+8$ that is partitioned into B=4 blocks of length L_b=2. The sub-figures demonstrate various stages of the algorithm. (**I**) presents the initialization of parallel sub-runs ('Run 1' and 'Run 2'). (**II**) shows the stage when the parallel runs are completed and all block-wise conditional results are available. (**III**) presents the stage when block 1 is merged with block 2 and block 3 is merged with block 4 and finally, (**IV**) shows the final stage when the all blocks are merged and conditional results are available for the whole section.

Similarly, the parallel merge step for the Viterbi recursions are given as

$$\delta_r^{\{b-1,b\}}(s_t \mid s_{\tau'_{b-2}}) \;=\; \max_q \left\{ \delta_q^b(s_t \mid s_{\tau'_{b-1}}=q)\;\; \delta_r^{b-1}(s_{\tau'_{b-1}}=q \mid s_{\tau'_{b-2}}=r) \right\} \tag{IV.35}$$

and for blocks b and $b{+}1$ for the backward pass

$$\beta_r^{\{b,b+1\}}(s_t \mid s_{\tau_{b+2}}) \;=\; \sum_q \left\{ \beta_q^b(s_t \mid s_{\tau_{b+1}}=q)\;\; \beta_r^{b+1}(s_{\tau_{b+1}}=q \mid s_{\tau_{b+2}}=r) \right\}, \tag{IV.36}$$

where r is the index of sub-runs on block $b{+}1$ (and refers to the condition on $s_{\tau_{b+2}}$), q is the index of sub-runs on block b (and refers to the condition on $s_{\tau_{b+1}}$).

The summation in these merge steps is over states at the intermediate block-boundary, and all $S{\times}S$ result matrices correspond to S conditions and S states.

Since these merge operations have to be performed for all states in all sub-runs (see Figure IV.5), the complexity is $\mathcal{O}(S^3)$ for each block of the stand-alone forward algorithm and $\mathcal{O}(L_b S^3)$ for a block of the forward-backward and Viterbi algorithms.

Possibly the most promising approach for merging blocks in parallel is in the form of a parallel reduction (as in Turin (1998) and Nielsen and Sand (2011), except that $L_b{>}1$ is allowed - see Section 2.2). The parallel reduction may be performed with a binary tree-based schedule

$$\Big(\big((b_1\, b_2)\; (b_3\, b_4) \big) \;\cdots\; (b_{2k-1}\, b_{2k}) \;\cdots \Big)$$

over blocks. The merge operations are performed for each block pair (e.g. b_1, b_2) at the leaf level of the tree. In each consecutive iteration, the merge operations are performed on pairs of the previous results, obtaining the results that correspond to the parent nodes in the tree (e.g. merging $b_{1:2}$ and $b_{3:4}$ to get $b_{1:4}$), until the root is reached with a single result for the whole observation.

This reduction requires $\log B$ iterations, leading to a parallel run time of $\mathcal{O}(S^3 \log B)$ for class I and $\mathcal{O}(L_b S^3)$ for class II algorithms, with a platform requirement $C \geq B$. Algorithm 13 presents a pseudocode for employing parallel merge.

Pseudocode 13 Complete pseudocode for a parallel Forward Algorithm with sequence partitioning using binary-tree reduction in the merge step, with block size $L_b = 1, \ \forall b \in \mathcal{B}$. This pseudocode corresponds to algorithm (III.A*).

parallel for all position $t \in \{1, .., T\}$ **do** ▷ the forward pass
  if $t = 1$ **then**
    Calculate $\alpha(s_1) = p(y_1|s_1)\, p(s_1)$ ▷ (IV.12)
  else
    Calculate $\alpha^{r_t}(\, s_t \,|\, s_{t-1}{=}r) \ = \ p(\, y_t \,|\, s_t)\, p(\, s_t \,|\, s_{t-1}{=}r_t)$ ▷ eq. (IV.15)
  end if
end parallel
parallel tree reduction over block pairings **do** ▷ the merge step
  for all $s_{\tau'_{b+1}} \in \mathcal{S}$ **do**
    for all $s_{\tau'_{b-1}} \in \mathcal{S}$ **do**
      Obtain $\alpha(s_{\tau'_{b+1}} \,|\, \alpha(s_{\tau'_{b-1}})$ from $\alpha^{r_{b+1}}(s_{\tau'_{b+1}} \,|\, s_{\tau'_b})$ and $\alpha^{r_b}(s_{\tau'_b} \,|\, s_{\tau'_{b-1}})$ ▷ eqs. (IV.34)
    end for
  end for
end tree reduction
Calculate the final $p(y_{1:T'})$ or $p(s_{T'}|y_{1:T'})$ as normally

4 Strategies, matrix formulation and performance

One of the aims in this chapter is to explore and prove the applicability and relevance of block parallelism – either on its own or in combination with other parallelisms. Since this is the first work on the topic, the focus will be to assess the optimal theoretical performance as a function of the parallelisation strategies employed. This section explores the possible combinations of parallelisms (these combinations will be referred to as 'strategies') applicable to HMM algorithms and assesses their applicability and optimality on various computation platforms.

First, a suitable parallel computation model is defined, with a performance analysis framework designed specifically for HMMs. Next, the space of possible strategies is explored, including their matrix formulations. Finally, the strategies are compared in terms of highest theoretically achievable performance.

4.1 Parallel computation model and performance analysis framework

In traditional algorithm analysis, applicability and performance are assessed in terms of computation and space complexity. As explained in Chapter III, execution times and the number of operations are not expected to be proportional in the case of parallel algorithms as assumed sequential algorithms. In fact, the very purpose of parallelisation is to reduce execution times for algorithms with a given number of operations. Therefore, it is necessary to treat parallel execution time and computation complexity separately. Further, in many

cases parallelisation itself may introduce extra computation as a cost of accelerated execution. These costs lower the theoretical maxium parallelisation efficiency ($\xi^{(W)}$) of strategies (W).

When the aim is to make comparisons between the performance of different parallelisation strategies, the achievable parallel accelerations ($X^{(W)}$) and the efficiencies ($\xi^{(W)} = X^{(W)}/D^{(W)}$, where D is the theoretically applied parallelism) are usually more informative than the overall execution times.

The performance of parallel algorithms depends on a multitude of factors in practice, including the parallelisms exploited, the implementation details and the hardware and software platforms available. When assessing the space, computation and time complexities of algorithms, it is sometimes useful to estimate them with relatively accurate functions of the problem size (e.g. input size), but more often the asymptotic behaviours are assessed as rate of growth $\mathcal{O}(\bullet)$ in practice. The benefits of the latter approach is that it focuses on higher order differences that are governed by algorithm design and omits constant factor differences that would otherwise depend on computing platforms, compilers, programming style and further details as well. For simplicity, the rate of growth is used in most of this section for assessing the requirements of the recursion and merge steps of parallel algorithms. When strategies are compared directly in the space of problem sizes (in terms of S, T), the asymptotic formulas are evaluated numerically with the assumption that all constant factors are equal to one. While no assumption on the constant factors is expected to hold in general, it is necessary to make such assumptions to allow for the visual assessment of the performance landscape of strategies (see Section 4.3).

In parallel programming, communications between threads can have a significant effect on performance. In standard sequential HMM algorithms, communications include setup time (loading $\boldsymbol{E}$, $\boldsymbol{A}$) reading in y_t, at every t (and maybe $\boldsymbol{E}_t$ and $\boldsymbol{A}_t$ for non-homogeneous models). During recursions, synchronisations also happen at every t (there is a computation barrier at every t), which includes communication between threads. There is a fixed amount of communication that has to be performed at every t. If there is a sufficient time (a sufficient number of compute operations on each thread) between these synchronisations and communication is fast, the communications costs may be close to negligent (hidden) on some platforms. If there is little time between synchronisations – for instance as a result of a high-degree of parallelism – and communications are slow, communications may dominate run time and may negate every attempt to achieve parallel acceleration.

4.1.1 The parallel computation model

When assessing the performance of parallel algorithms, one can assume infinite space, infinite possible parallelism and instant communications, but assessments in such a framework would

be overly limited. The alternative is to model platform constraints and perform evaluations with such platform models. Both avenues are explored in this work.

In the parallel computing literature the two most widely used parallel platform models are the parallel random access machine (PRAM) and the bulk synchronous parallel (BSP) model (Fortune and Wyllie, 1978) (Valiant, 1990). The PRAM model is an extension of the standard random access machine (RAM) model. The PRAM model provides a constraint on the achievable parallelism (C) but assumes shared memory between threads and disregards communication and synchronisation costs, while the more complicated BSP model allows for a detailed assessment of communication and synchronisation costs when evaluating algorithm complexity.

Since the communication and synchronisation costs are increasingly dominating computation times in modern parallel architectures, it is safe to assume that the PRAM model is insufficient for assessing th performance of parallel algorithms. On the other hand, the BSP model requires detailed assumptions to be made on model parameters, which may be hard to determine for existing platforms and which could also lead to loss of generality. Indeed, when specific algorithms are assessed on a given computer platform, it may be beneficial to use an accurate model with finely tuned parameters, but since the focus was to explore performance more generally on a wide scale of platforms, a middle ground had to be found. A new programming model – referred to as the simplified communications model (SCM) – is defined here for use in this section.

In SCM C is assumed to be known, the memory available for threads is unlimited and the communication and synchronisation costs (Q) are assessed qualitatively, using four general grades $Q \in \{0, L, M, H\}$, which stand for 'effectively zero', 'low', 'medium' and 'high' (see Table IV.1). The C limit comes into effect when it is smaller than the algorithmically achievable parallelism (D). In such cases, the bounded theoretical run time (ϱ), which assumes that the effective parallelism ($P = \min(C, D)$) is less than the unbounded theoretical run time (ρ), where it is assumed that the effective parallelism is equal to the theoretical (algorithmic) parallelism ($P = D$). The platform parameter C can be thought of as a resource that the various prallelisation strategies can utilise to achieve acceleration.

The effects of communication requirements are considered by assuming

$$\varrho^{(W)} = \begin{cases} \infty & \text{when} \quad Q^{(platform)} < Q^{(W)} \\ \text{unaffected} & \text{when} \quad Q^{(platform)} \geq Q^{(W)} \end{cases} \tag{IV.37}$$

where $\varrho^{(W)}$ is the constrained execution time of strategy W, while $Q^{(platform)}$ denotes the

Table IV.1: Communication requirements for algorithms and communication channels (of various platforms) under SCM.

Q	Algorithms	Platforms
0 (zero)	The algorithm requires minimal communication between threads, e.g. only at the end of a run.	Distributed platforms with remote nodes (e.g. all communications between threads is over the internet).
L (low)	The algorithm requires infrequent communications between threads.	Threads run on separate nodes, which are connected at least into a local area network, e.g. classic compute clusters.
M (medium)	The algorithm requires frequent communications between threads.	Multiple threads run on one chip, e.g. multi-core systems, such as CPUs in modern single-node compute servers.
H (high)	The algorithm requires frequent parallel reductions between threads.	The platform supports parallel reductions at a hardware level, e.g. modern GPGPUs.

Note: While it is usually trivial to distinguish Q=0 and Q=H for algorithms (and relatively easy for existing platforms), the distinction between Q=L and Q=M may be dependent on S and T and possibly on some further assumptions.

highest level[7] of communication supportable by the platform, and $Q^{(W)}$ represents the degree of communication requirements of strategy W. This simplification does not fall far from the view of a programmer that aims to hide communication costs behind sufficient amount of computation during practical implementations. SCM may be seen as a simplified version of the BSP model, assuming communication costs to be infinite or null, dependent on synchronisation frequency (the time between barriers). Also, an SCM platform with $Q = H$ is essentially equivalent to a PRAM model.

When employing this model, a crucial step is to determine $Q^{(W)}$ for strategies by assigning them into one of the four SCM communication-complexity classes. For parallel HMM algorithms, the overall Q of strategy W is given by the maximum Q required by the recursion and merge steps applied.

While real-world platforms may have multiple different communication channels (e.g. in a system of distributed GPU clusters), SCM assumes a single type of communication channel for a platform for simplicity. The aim with this model is to be able to distinguish general types of computing platforms without having to make further assumptions on their capabilities.

[7]The level of communication may be thought of as a collective quality based on the frequency and speed of synchronisations, memory access times and data transfer times between parallel threads.

4.1.2 Performance evaluation for HMM algorithms

The most direct way of assessing the performance of parallelisation strategies is to compare their highest achievable parallel accelerations (χ^*). Similar to the unbounded and the constrained execution times (ρ and ϱ), the χ^* are expected to be dependent on the problem size (S and T in HMM algorithms) for most strategies. The acceleration is given as the rate at which execution time is shorter compared to standard sequential execution

$$\chi^{*(W)}(S,T) \;=\; \frac{\varrho^{(seq)}(S,T)}{\varrho^{*(W)}(S,T)} \;=\; \frac{S^2T}{\varrho^{*(W)}(S,T)}, \tag{IV.38}$$

where

$$\varrho^{*(W)}(S,T) = \varrho^{*(rec)}(S,T) + \varrho^{*(merge)}(S,T) \tag{IV.39}$$

is the estimated optimal constrained parallel execution time. The $\varrho^{*(W)}$ are given based on the unbounded theoretical execution time $\rho^{*(W)}$ (it is C that is assumed unbounded, not the execution time itself) and the C constraint of a given platform model.

In the case of block-parallel strategies, the optimal execution times

$$\varrho^{*(W)}(S,T) \;=\; \max_{(L,B)} \left\{\varrho^{(W)}(S,T)\right\} \tag{IV.40}$$

can be found once the optimal parallelisation parameters

$$\theta^{*(W)}(S,T) \;=\; \Big(L^*(S,T),\; B^*(S,T) \Big) \;=\; \arg\max_{(L,B)} \left\{\varrho^{(W)}(S,T)\right\} \qquad L,B \in \mathbb{N}, \quad L,B \geq 2 \tag{IV.41}$$

are known. L is the length of the calculation section for which the algorithm is performed at a given time ($2 < L < T$) and B is the the number of blocks used in block-parallelism ($2 < B < L$).

It is important to note that only strategies for class II algorithms (algorithms, where results are calculated for all t) assuming $L_b > 1$ require optimisation over L and B jointly. Strategies that assume $L_b = 1$ (and $B = L$) can be optimised over L only and strategies for class I algorithms (algorithms, where results are calculated for $t = T$ only) assuming $L_b > 1$ have to be optimised over B only.

Parallel reductions

In the realm of parallel algorithms it is often required to perform a reduction operation (e.g. sum or max) in parallel. These reductions can not be fully parallelised and often introduce extra communication costs. Since most parallel reductions are performed following a binary tree schedule, they can be assumed to require $\mathcal{O}(\log N)$ iterations for N original elements (there is one iteration for each level of the tree). Although the computation cost and run time increase introduced by parallel reductions is often not $\log N$, the rate of growth $\mathcal{O}(\log N)$ is a good approximation and will be used in all performance analyses. Wherever χ^* were actually calculated, $\log x = \log_2 x$ was used.

Complex parallel algorithms

Many times the parallelisation of algorithms requires some extra computation (for instance, a merge step). This extra computation is the cost of the parallelism achieved and has to be accounted for in the performance analysis.

In order to do so, the execution time performance of parallel HMM algorithms can be expressed as the unbounded theoretical run time

$$\begin{aligned} \rho^{(W)} &= \rho^{(recursion)} + \rho^{(merge)} \\ &= \frac{\rho^{(seq)}}{D_1^{(W)}} Z_1^{(W)} + \frac{Z_2^{(W)}}{D_2^{(W)}} Z_3^{(W)} \\ &= \frac{S^2 T}{D_1^{(W)}} Z_1^{(W)} + \frac{Z_2^{(W)}}{D_2^{(W)}} Z_3^{(W)} \end{aligned} \tag{IV.42}$$

and, similarly, the expected constrained run time is

$$\varrho^{(W)} = \frac{S^2 T}{P_1^{(W)}} Z_1^{(W)} + \frac{Z_2^{(W)}}{P_2^{(W)}} Z_3^{(W)}, \tag{IV.43}$$

where the first term refers to the run time of the recursion and the second term refers to the merge step (where applicable); ρ_{seq} is the original sequential complexity, $P_1^W = \min\{D_1, C\}$ is the practical parallelism achievable for the recursion part and $P_2^W = \min\{D_2, C\}$ for the merge step; Z_1^W refers to the computation cost directly incurred by parallelising the recursions, Z_2^W is the base computational cost of the merge step and Z_3^W is the cost of applying parallelisms for the merge step.

The ρ are given for the parallel recursion and merge steps of various strategies in Section 4.2 and the ϱ and χ^* are evaluated and compared in Section 4.3 under various SCM platform models.

4.2 The arsenal of HMM parallelisation strategies

This section explores possible combinations of parallelisms (strategies) for the dynamic programming recursions of HMM algorithms. Most explanations assume the forward algorithm as an example (both in its class I and class II form), but all methods are applicable to the backward and Viterbi algorithms as well, with trivial modifications following the derivations in Section 3.

4.2.1 Standard HMM parallelisms

Before analysing the various strategic possibilities in block-parallelism, the performance assessment framework will be introduced through an overview of standard HMM parallelisation schemes.

Trivial HMM parallelisms

The most obvious parallelisms are those over observation sequences and different parametrisations (G different values of $\boldsymbol{\lambda} = (\boldsymbol{A}, \boldsymbol{E}, \boldsymbol{\pi})$). Algorithms may be run in parallel on any number $1 \leq K' \leq K$ of observation sequences and any number of parametrisations $1 \leq G' \leq G$ in parallel, while performing calculations across K' and G' bundles sequentially. In these cases calculations across K' and G' re inherently independent, hence parallelising them is trivial. With G' parametrisation and K' observations in a bundle, these parallelisms may translate into a close to $D = G' \times K'$-fold acceleration. The platform requirements are $C \geq D = K'G'$ and $Q \geq 0$. The achievable acceleration is $\chi = \mathcal{O}(G'K')$ and the acceleration efficiency is $\xi = \mathcal{O}(1)$.

Parallelisms over states

HMM algorithms may be parallelised 'twice' over the state-space. First, at each time t, calculations for each 'present' state $s_t \in \mathcal{S}$ are independent and hence relatively easily parallelisable. The cost of this parallelisation is that threads performing calculations for different s_t have to synchronise results before calculation can begin for $t+1$. It is expected that this will lead to relatively frequent synchronisations and a corresponding $Q^{(S.S)} \geq M$ communication requirement. The highest achievable parallelism is $D^{(S.S)} = S$ and the highest achievable

acceleration is $\chi^{S.S} = \mathcal{O}(S)$ and the highest acceleration efficiency is $\xi^{(S.S)} = \mathcal{O}(1)$.

The second possibility for parallelism is at the level of summations over the 'previous states' s_{t-1} and will be coded as S.S'. The summations require a parallel reduction, hence they have a $Z_1^{(S.S')} = \mathcal{O}(\log S)$ extra parallel computation cost. The reductions are also most efficient when the available platform allows for efficient reductions, and, hence have a $Q^{(S.S')} = H$ communication requirement. The limit of parallelism is $D^{(S.S')} = \mathcal{O}(S)$, the limit of acceleration is $\chi^{(S.S')} = \mathcal{O}(S/\log S)$ and the acceleration efficiency bound is $\xi^{(S.S')} = \mathcal{O}(1/\log S)$.

Because of the higher requirements and lower efficiency of S.S' it is clear that one would apply S.S before applying S.S' when C is limited, but the two parallelisms may also be applied together (coded as S.SS') when C is sufficiently larger than S. In such cases, the parallelism limit is $D^{(S.SS')} = \mathcal{O}(S^2)$, the acceleration limit is $D^{(S.SS')} = \mathcal{O}(S^2/\log S)$ and the efficiency bound is $\xi^{(S.SS')} = \mathcal{O}(1/\log S)$. Table IV.2 summarises the characteristics of standard HMM parallelisms over the state space.

Table IV.2: Standard HMM algorithm parallelisms over the state space.

Strategy (W)	Parallelisms over	Parallelisms and costs			$\mathcal{O}(\bullet)$ performance, $C = \infty$			Q
		D_1	Z_1	Z_2	ρ	χ	ξ	
S.S	Present states s_t	S	-	-	SL	S	1	M
S.S'	Previous states s_{t-1}	S	$\log S$	-	$SL \log S$	$S/\log S$	$1/\log S$	H
S.SS'	s_t, s_{t-1}	S^2	$\log S$	-	$L \log S$	$S^2/\log S$	$1/\log S$	H

D is the algorithmically achievable highest parallelism, Z are additional computational costs of parallelisation, ρ is the unconstrained parallel run time, χ is the highest achievable parallel acceleration and ξ is the corresponding efficiency. Q refers to communication and synchronisation costs and can be null (0), low (L), medium (M) or high (H).

The formulas in the table are the theoretical expectations. In practice, D, ρ, χ and ξ can be limited by C, actual communication costs and many other factors.

Strategy S.S' (grey) should be avoided on its own due to its low efficiency, but the corresponding parallelism may be applied in S.SS' when C is sufficiently larger than S.

The optimality of standard HMM parallelisarion strategies

Due to the differences in efficiency, implementation complexity and platform dependence between standard HMM parallelisation strategies, parallelism over K and G should be used first if applicable, then parallelism over present states if $C > KG$ and $Q \geq M$. Finally, if there is still sufficient possible parallelism available ($C > KGS$) and the available platform

allows for fast reductions ($Q^{(platform)} = H$), parallelism over past states may be included as well for time-critical applications. As it will be demonstrated, choosing the optimal strategy is significantly more complicated when parallelism over blocks is considered.

Due to the triviality of parallelisms over K and G, they will not be discussed further, but it is important to keep in mind that these parallelisms will use up some of the parallelism available when combined with others. Hence, for applications where more than one observation or more than one parametrisations are expected, the available parallelism may be thought of as $C_{eff} = C/(G'K')$ in later parts of this section, but will be written as C for simplicity. It is also worth noting that on real computing platforms, where connectivity may be different on different levels of nodes, it is often trivial to distribute computation in a way that the parallel threads corresponding to parallelisms over K and G are spread across less-connected super-nodes (e.g. remote centres or remote clusters), while tightly coupled computing units (nodes, cores, etc.) can work on sub-tasks (e.g. calculations for a single observation) where parallelisms require more efficient communications.

4.2.2 Strategies including block-parallelism

The focus of this work is on HMM algorithm parallelisation including block-parallelism. For an intuitive, detailed explanation of block-parallelism with sequential and parallel merge, see Section 3. Here we will explore the space of possible parallelism combinations for the recursion and merge steps (these will be the recursion and merge strategies).

As explained in Section 3, block-parallelism can be separated into two stages, a recursion and a merge step. Since it is possible to separate the recursion and merge operations, it is possible to use the full computing power of our platforms for both steps if a number of conditions are met. One of these conditions is that there is sufficient memory to hold the results from the recursion step, and the second is that the threads used in the merge step can access all required results from the recursion step without significant performance loss. For all subsequent analysis, it will be assumed that the first condition is met – it is always possible to choose L to be small enough to fit all results in memory, although limiting L for this purpose may have severe performance implications practice, which will not be explored here. The second condition is met if the platform uses a single memory space accessible by all threads (such as GMEM or SMEM on a single CUDA GPU) and allows for fast access. Alternatively, either (a potentially large amount of) data transfers are required after the recursion step (e.g. between different remote nodes in the case of distributed computing) or implementations have to be constrained to 'sufficiently similar' parallelisms (and thread assignments) in the recursion and merge steps so that data transfers can be avoided. Deciding what is sufficiently similar on a platform will be dependent on the memory architecture and the specific parallel implementation.

In order to explore the whole space of possible strategies, unlimited memory size and global memory access will be assumed for all threads. In the light of these assumptions, the details of recursion and merge strategies will be discussed separately in this section, followed by a performance analysis of complete strategies (i.e. including compatible recursion plus merge steps) in Section 4.3. When applying block-parallelism, the recursion and merge strategies have to be chosen as a compatible pair, with identical assumptions on L_b. The hope is that this section will provide some background for understanding these strategies, while the theoretical performance analysis results in Section 4.3 will give some guidance for choosing the most suitable strategies for real applications.

In the explanations, it is assumed that the length of observations is divided into n pieces of L-long sections. Computation on these sections is performed consecutively, while computation inside these sections is performed with parallelisms. The significance of using L is that it can be chosen to optimise parallel performance and/or to be able to reduce memory requirements compared to cases when the algorithm is run on the whole observation length - in the following discussions $L<T$ cases are considered exclusively for optimisation reasons.

4.2.3 The parallel recursion step

The aim in the parallel recursion step is to produce the conditional forward messages $\alpha_r^{(b)}(\, s_{\tau_b} \,|\, s_{\tau'_{b-1}} = r)$ for all blocks $b > 1$. Since the initial distribution $\boldsymbol{\pi}$ is available for the first block, it is expected that the standard (unconditional) forward message results are produced for the first block with standard HMM calculations and are available. Parallelisms may be applied on the first block, but this option and its effect will not be reviewed here. Without loss of generality, it may be assumed that all following explanations and analyses correspond to blocks $b \in \{2, ..., B\}$.

Recursion strategies can be divided into two qualitatively different groups according to assumptions made on the length of blocks. The first group of strategies will assume the special case $L_b=1$ and the second group will assume $L_b>1$. The characteristics of strategies falling into these groups will be discussed separately.

Assuming $\mathbf{L_b=1}$

A special case for block-parallelism is when the length of blocks is assumed to be a single piece of observation. This case was introduced by Turin (1998) - see Section 2 for details. The calculations in the recursion step are essentially $\mathbf{P}(y_t) = \mathbf{A}\mathbf{E}^{\mathbf{d}}(y_t)$, which is equivalent to the calculation of

$$\alpha_r^{(b)}(\, s_t \,|\, s_{t-1}) \;=\; \alpha_r^{(b)}(\, s_{\tau_b} \,|\, s_{\tau'_{b-1}}{=}r) \;=\; \mathrm{P}(\, y_{\tau_b} \,|\, s_{\tau_b})\, \mathrm{P}(\, s_{\tau_b} \,|\, s_{\tau'_{b-1}}{=}r) \tag{IV.44}$$

for sub-run initialisations, as introduced in Section 3 as equation (IV.14). This is basically a normal block-parallel approach, but only the block initialisations have to be performed for every block, when $L_b = 1$. The benefit of this approach is that the number of calculations required remains $\mathcal{O}(S^2)$ for each t (observing that $\mathbf{E}^{\mathbf{d}}(y_t)$ is a diagonal matrix). The disadvantage is that each piece of observation requires a thread ($C \geq D_1 = L$), which may be sub-optimal when C is constrained.

In its basic form, the only parallelism is over the section length L (this strategy will be coded as R0), but there are further possible parallelisms that are easy to identify from either formulations. One possibility is to calculate the $\alpha_r^{(b)}(\, s_{\tau_b} \,|\, s_{\tau'_{b-1}}{=}r)$ in parallel for all sub-runs r, which corresponds to parallelising multiplications over the rows of $\mathbf{P}(y_t)$ (this strategy will be coded as R0.R). Alternatively, it is possible to calculate the matrix entries parallel for all s_{τ_b}, which corresponds to the columns of $\mathbf{P}(y_t)$ (strategy R0.S), or one can employ both parallelisms (strategy R0.RS).

It is easy to see that all calculations are independent for all recursion strategies with $L_b{=}1$, hence $Q{=}0$. Although the efficiency of calculating $\mathbf{P}(y_t)$ may be $\mathcal{O}(1)$, the overall efficiency may be lower when including the merge step. The lowest achievable execution times for the recursion are $\mathcal{O}(nS^2)$, $\mathcal{O}(nS)$, $\mathcal{O}(nS)$ and $\mathcal{O}(n)$ for strategies R0, R0.R, R0.S and R0.RS, respectively[8]. The space requirement is $\mathcal{O}(S^2L)$ for all recursion strategies with $L_b = 1$. Table IV.3 summarises the characteristics of all parallel recursions.

Assuming $\mathbf{L_b > 1}$

In the more general setting, the parallel sub-runs are performed for more than one observation with

$$\alpha_r^{(b)}(s_t \,|\, s_{\tau'_{b-1}}{=}r) \;=\; \mathrm{P}(\, y_t \,|\, s_t) \sum_{s_{t-1}} \mathrm{P}(\, s_t \,|\, s_{t-1})\;\; \alpha_r^{(b)}(\, s_{t-1} \,|\, s_{\tau'_{b-1}}{=}r) \tag{IV.45}$$

calculated for $t \in \{\tau_b{+}1, .., \tau'_b\}$ in each block $b \in \{1, ..., B\}$ of length $L_b{=}L/B$. This requires $\mathcal{O}(n{\times}L{\times}S{\times}S^2)$ operations, which can be performed in $\mathcal{O}(TS^3/B)$ time with $D_1{=}B$-fold parallelism (this basic version is coded R1). Clearly, the disadvantage of this approach is the $\mathcal{O}(S)$-fold increase in the number of required operations and space requirement, while the advantage is that L and B may be chosen more flexibly, which will have great effects on

[8] As a reminder, $n{=}\lceil T/L \rceil$ and $T, n, L \in \mathbb{N}$ and $n{=}T/L$ are assumed for simplicity.

Table IV.3: Parallelisation possibilities for the recursion step when including block-parallelisms.

Strategy (W)	Parallelisms over	Parallelisms and costs		$\mathcal{O}(\bullet)$ recursion performance, $C=\infty$			Q
		D_1	Z_1	$\rho^{(rec)}$	$\chi^{(rec)}$	$\xi^{(rec)}$	
When $L_b=1$ is fixed							
R0	time t	L	-	nS^2	L	1	0
R0.R	t, sub-runs r	LS	-	nS	LS	1	0
R0.S	t, present states s_t	LS	-	nS	LS	1	0
R0.RS	t, r, s_t	LS^2	-	n	LS^2	1	0
For long blocks $L_b>1$							
R1	blocks b	B	S	nL_bS^3	B/S	$1/S$	0
R1.R	b, sub-runs r	BS	S	nL_bS^2	B	$1/S$	0
R1.S	b, present states s_t	BS	S	nL_bS^2	B	$1/S$	M
R1.S'	b, previous states s_{t-1}	BS	$S\log S$	$nL_bS^2\log S$	$B/\log S$	$1/(S\log S)$	H
R1.RS	b, r, s_t	BS^2	S	nL_bS	BS	$1/S$	M
R1.RSS'	b, r, s_t, s_{t-1}	BS^3	$S\log S$	$nL_b\log S$	$BS^2/\log S$	$1/(S\log S)$	H

Here $\rho^{(rec)}$, $\chi^{(rec)}$ and $\xi^{(rec)}$ correspond to the recursion calculations - when the merge step is also considered, run time is expected to be longer and acceleration and efficiency to be lower. Q refers to communication and synchronisation costs and can be null (0), low (L), medium (M) or high (H).

$n=\lceil T/L\rceil$ is the number of sections the calculation is split into, $nL_b\sim T/B$.

Block-parallelisations assume a merge step, the corresponding Z_2 is omitted from the table - it is $B\times S^2$ for class I and LS^2 for class II algorithms.

The formulas in the table are the theoretical optima. In practice, P_1 and $\varrho^{(rec)}$ will be used instead of D_1 and $\rho^{(rec)}$, and ξ will also be affected by C correspondingly.

Strategies R1.S' (grey), R1.RS' and R1.SS' should be avoided due to low efficiency, but parallelism over past states may be applied when when C is large and all other options for parallelism are exhausted.

performance, as shown in Section 4.3.

An equivalent matrix formulation for the forward algorithm with conditional sub-runs (for block b) may simply be

$$\boldsymbol{\alpha}_{\boldsymbol{t}}^{(\boldsymbol{b})} \;=\; \mathbf{I} \prod_{i=\tau_b}^{t<\tau'_b} \mathbf{P}(y_i), \tag{IV.46}$$

where the identity matrix represents the $s_{\tau'_{b-1}} = r$ conditions and $\boldsymbol{\alpha}_{\mathbf{t}}^{(\mathbf{b})}$ is a matrix of $\alpha_r^{(b)}$ results, with (i,j) coordinates being $\alpha_r^{(b)}(s_t{=}j \mid s_{\tau'_{b-1}}{=}i)$.

Just as in the $L_b{=}1$ case, further parallelisms may be applied on these recursions. Options include parallelisms over the r sub-runs (strategy R1.R), or standard state-wise parallelisms over previous and present states (strategies $R1.S$ and $R1.S'$) or various combinations (strategies R1.RS and R1.RSS'). Due to the reduced efficiency and the $Q{=}H$ communication requirements, parallelisms over the summation (past states s_{t-1}) should not be used unless all other parallelisms are already employed and the computation power of available platforms is still not fully utilised ($C{>}D_1{=}BS^2$).

The communication cost for R1 is $Q{=}0$, trivially. It is easy to see that the calculations on sub-runs can be performed independently, hence the communication cost for R1.R is also $Q = 0$. It is possible to use R1.S instead of R1.R if it is practical, but R1.S is expected to require $Q \geq M$, as in state-wise parallelisms. Strategies that include parallelisms over the summations require $Q = H$ and R1.RS is expected to require $Q \geq M$ due to frequent synchronisations.

The achievable lowest execution time ranges from $\mathcal{O}(TS^3/B)$ down to $\mathcal{O}(T \log S/B)$ and the acceleration ranges from $\mathcal{O}(B/S)$ to $\mathcal{O}(BS^2/\log S)$ (see Table IV.3).

4.2.4 The merge step

For the merge step it is assumed that the parallel recursions are completed (at least for the current L-long section), and all of the parallel computation power C is fully available[9]. Consequently, it is sensible to apply parallelisms to the merge step in order to improve overall acceleration, which will be discussed here.

Before exploring various strategy options for the merge step, it is also important to note that there are three distinct cases to be considered. The first case is when $L_b{=}1$ is assumed in the recursion step, irrespective of algorithm class. When $L_b{>}1$, parallelisms are expected to work differently for the two algorithm classes[10]. Table IV.3 summarises all sequential merge

[9] It may be possible to overlap the two stages and employ suitable optimisations, but that case will not be covered here.

[10] As a reminder: class I refers to algorithms that calculate final results only for T, while class II algorithms result are important for all $t \in \{1, ..., T\}$.

strategies.

4.2.5 Sequential merge

In every step of sequential merge, the conditional results ($\alpha_r^{(b)}(s_t \mid s_{\tau'_{b-1}}),\ t \in \{\tau_b, ..., \tau'_b\}$) of blocks $b>1$ are 'transformed' into normal unconditional results using the already calculated unconditional results corresponding to blocks $\{1, ..., b-1\}$.

The unconditional $\alpha(s_t)$ messages are calculated according to

$$\alpha(s_t) \;=\; \sum_r \alpha_r^{(b)}(s_t \mid s_{\tau'_{b-1}} = r)\ \ \alpha(s_{\tau'_{b-1}} = r) \tag{IV.47}$$

as introduced in Section 3.1.

Sequential merge assuming $\mathbf{L_b = 1}$ for class I and class II algorithms

Applying sequential merge with $L_b = 1$ is equivalent to performing the vector-matrix multiplications in $\boldsymbol{\pi} \prod_{t=1}^{T} \mathbf{P}(y_t)$ sequentially as $\boldsymbol{\alpha}_{\tau'_b} = \boldsymbol{\alpha}_{\tau'_{b-1}} \mathbf{P}(y_{\tau_b:\tau'_b})$ (see Section 2 for relevant prior works using the matrix formulation). This basic sequential merge for this case is coded as MS0. There need to be $nBL_b = T$ merge steps performed for a whole observation sequence, each with $\mathcal{O}(S^2)$ calculations. In most cases this will also mean an $\mathcal{O}(S^2T)$ overall runtime, with very limited acceleration (if at all), irrespective of the recursion strategy used.

Observing equation (IV.47), it is clear that there are options for S-fold parallelisms over $s_{\tau'_b}$ and r or an S^2 parallelism over both (these will be coded as MS0.S, MS0.R and MS0.RS, respectively). Because of the necessity of parallel reductions, parallelisms over r introduce an extra $\mathcal{O}(\log S)$ computation, a corresponding decrease in efficiency and a $Q = H$ communication cost. The communication cost for MS0.S is expected to be medium level, because synchronisations are required about every $\mathcal{O}(S)$ operations.

In terms of optimality, it is worth noting that MS0 only requires $Q = 0$ and that MS0.SR may achieve the highest acceleration when C is sufficient. The space requirement is $\mathcal{O}(S^2L)$ for class I and $\mathcal{O}(S^2L + ST)$ for class II algorithms. Strategies have to be optimised over $L = B$.

Table IV.4: Possible parallelisms for sequential merge.

Strategy (W)	Parallelisms over	Parallelisms and costs			$\mathcal{O}(\bullet)$ merge performance, $C=\infty$			Q
		Z_2	D_2	Z_3	$\rho^{(merge)}$	$\chi^{(merge)}$	$\xi^{(merge)}$	
For all HMM algorithms with $L_b=1$								
MS0	-	$nL\times S^2$	-	-	$nL\times S^2$	-	-	0
MS0.S	present states s_t	$nL\times S^2$	S	-	$nL\times S$	S	1	M
MS0.R	sub-runs r	$nL\times S^2$	S	$\log S$	$nL\times S\log S$	$S/\log S$	$1/\log S$	H
MS0.SR	r, s_t	$nL\times S^2$	S^2	$\log S$	$nL\times\log S$	$S^2/\log S$	$1/\log S$	H
For class I algorithms with $L_b>1$								
MS1	-	$nB\times S^2$	-	-	$nB\times S^2$	-	-	0
MS1.S	present states s_t	$nB\times S^2$	S	-	$nB\times S$	S	1	M
MS1.R	sub-runs r	$nB\times S^2$	S	$\log S$	$nB\times S\log S$	$S/\log S$	$1/\log S$	H
MS1.SR	r, s_t	$nB\times S^2$	S^2	$\log S$	$nB\times\log S$	$S^2/\log S$	$1/\log S$	H
For class II algorithms with $L_b>1$								
MS2	-	$nB\times L_bS^2$	-	-	$nB\times L_bS^2$	-	-	0
MS2.S	present states s_t	$nB\times L_bS^2$	S	-	$nB\times L_bS$	S	1	L/M
MS2.T	block length t	$nB\times L_bS^2$	L_b	-	$nB\times S^2$	L_b	1	L/M
MS2.R	sub-runs r	$nB\times L_bS^2$	S	$\log S$	$nB\times L_bS\log S$	$S/\log S$	$1/\log S$	H
MS2.ST	s_t, t	$nB\times L_bS^2$	L_bS	-	$nB\times S$	L_bS	1	M
MS2.STR	s_t, t, r	$nB\times L_bS^2$	L_bS^2	$\log S$	$nB\times\log S$	$L_bS^2/\log S$	$1/\log S$	H

Here $\rho^{(merge)}$, $\chi^{(merge)}$ and $\xi^{(merge)}$ correspond to the merge step - run time is expected to be longer and acceleration and efficiency to be lower with the recursion step. $n=\lceil T/L\rceil$ and $nL_b\sim T/B$ as before.

The formulas in the table are the theoretical optima. In practice, P_2 and $\varrho^{(merge)}$ will be used instead of D_2 and $\rho^{(merge)}$, and $\xi^{(merge)}$ will also be affected by C correspondingly.

Strategies MS0.R, MS1.R and MS2.R (all grey) should be avoided on their own due to their low efficiency and high communication costs, but may be applied together with other parallelisms when C is sufficiently large.

Sequential merge assuming $\mathbf{L_b>1}$ for class I algorithms

For class I algorithms, sequential merge with $L_b > 1$ also follows equation (IV.47), but merge has to be performed only for every $t = \tau'_b$, which may be thought of as calculating $\boldsymbol{\alpha}_{\tau'_b} = \boldsymbol{\alpha}_{\tau'_{b-1}}\mathbf{P}(y_{\tau_b:\tau'_b})$ with a single vector-matrix multiplication for each block. The result

may be a considerable ($\times\mathcal{O}(L/B)$) performance improvement and space reduction compared to the $L_b=1$ case.

The possible further parallelisms are identical to those for sequential merge with $L_b = 1$, as well as the Q requirements and relative performance differences (the strategy codes are MS1, MS1.S, MS1.R and MS1.SR, similarly to the $L_b=1$ case). The differences compared to the $L_b=1$ case are that recursions with $L_b>1$ are required, there is a $\times\mathcal{O}(L/B)$ speedup, the space requirements are $\mathcal{O}(S^2B)$ and optimisations may be performed over B and not L ($L=T$ is optimal in theory, but may be constrained by the limited space available in practice, which is not covered here).

Sequential merge assuming $\mathbf{L_b>1}$ for class II algorithms

In the case of class II algorithms, merge has to be performed for all t in each block, not just for the end $t=\tau'_b$ of each block. This is equivalent to calculating $\boldsymbol{\alpha}_{\tau_b\leq t\leq\tau'_b} = \boldsymbol{\alpha}_{\tau_b-1}\mathbf{P}(y_{\tau_b:t})$ vector for every t in every block.

The increased number of calculations (compared to the class I case) have significant performance and optimality implications, but also introduce an additional possible parallelism over the block length, since the calculations of all $\alpha(s_t)$ are independent, given $\alpha_r^{(b)}(s_t \mid s_{\tau'_{b-1}}=r)$ and $\alpha(s_{\tau'_{b-1}}=r)$.

The not-parallelised version of sequential merge step is MS2, the strategy with parallelism over s_t is MS2.S, the one with parallelism over r is MS2.R, the one with parallelism over block length is MS2.T and the most sensible combined strategies are MS2.ST and MS2.STR. Strategies that include parallelisms over r introduce an efficiency decrease as well as high communication costs ($Q=H$), MS2.ST is expected to require frequent synchronisations ($Q=M$) and the communication costs of MS2.S and MS2.T are $Q\in\{L,M\}$, dependent on problem size and specific platform assumptions.

Optimality is particularly complex to determine and is, again, highly dependent on both problem size and the target platform, but MS2 is noteworthy for its low communication requirement and MS2.STR for the highest theoretically achievable acceleration on sufficiently powerful platforms.

The space requirements are $\mathcal{O}(B\times L_bS^2)=\mathcal{O}(LS^2)$ for both the merge and the corresponding recursion steps and $\mathcal{O}(LS^2+TS)$ overall, for all strategies. Strategies have to be optimised over both L and B simultaneously.

4.2.6 Parallel merge

In general, parallel merge takes two sets of conditional messages corresponding to two consecutive blocks and produces a single set of conditional messages for the concatenated block as

$$\alpha_r^{\{b-1,b\}}(s_t \,|\, s_{\tau'_{b-2}}) \;=\; \sum_q \; \Big\{ \; \alpha_q^b \,(s_t \mid s_{\tau'_{b-1}} = q) \;\; \alpha_r^{b-1} \,(s_{\tau'_{b-1}} = q \mid s_{\tau'_{b-2}} = r) \; \Big\}. \tag{IV.48}$$

This is equivalent to performing the matrix-matrix multiplications

$$\mathbf{P}(y_{\tau_{b-1}:\tau'_b}) \;=\; \mathbf{P}(y_{\tau_{b-1}:\tau'_{b-1}}) \, \mathbf{P}(y_{\tau_b:\tau'_b}), \tag{IV.49}$$

where the rows correspond to sub-runs and the columns correspond to states in both input and result matrices.

In parallel merge, pairs of neighbouring blocks can be concatenated in any order. One expectedly efficient approach is to perform parallel merge in the form of a tree-reduction over all blocks (see 2 for previous works on this tree-reduction-based parallel merge). In this section, this tree-reduction-based parallel merge will be assumed to be used in every application of parallel merge, which implies $\mathcal{O}(S^3)$ operations to be performed in parallel over all blocks, for every τ'_b or t required, for $\mathcal{O}(\log B)$ levels of the reduction tree. When a merge operation includes $t=1$, unconditional results may be calculated according to (IV.47), which is somewhat faster (this performance difference will be disregarded for simplicity). The last step in parallel merge (corresponding to the root of the tree) includes the whole section and produces the final unconditional results for the whole section.

When assessing applicability, it is good to keep in mind that all parallel merge is expected to require at least $Q \geq L$ communications due to the synchronisations after calculations corresponding to every level of the tree. It is intuitive to think that sequential merge has large computation costs ($Z_3 \geq \mathcal{O}(S \log L)$, see Table IV.5), but detailed analyses show that parallel merge may have higher relevance in terms of performance than expected (see results in Section 4.3).

Similarly to sequential merge, the distinction between the three cases (case $L_b = 1$, case $L_b > 1$ for class I algorithms and case $L_b > 1$ for class II algorithms) is equally important for parallel merge and, hence, these will be discussed separately.

Parallel merge assuming $\mathbf{L_b = 1}$ for class I and class II algorithms

When the parallel tree-reduction is performed over every single individual observation and without further parallelisms (MP0), it may execute in $\mathcal{O}(nS^3 \log L)$ time when sufficient parallelism ($C \geq D = L$) is available.

It is clear that parallel merge may be further parallelised over the present states s_t, sub runs r and q and any combinations of these. The most relevant strategies are MP0.S, MP0.R, MP0.Q, MP0.SR and MP0.SRQ (see Table IV.5). Because of parallel summations, strategy MP0.Q is sub-optimal, as expected, but parallelisms over Q may be included when all other options are exhausted and the C limit is still not reached (as in MP0.SRQ).

The space requirement is $\mathcal{O}(S^2 L)$ for class I, and $\mathcal{O}(S^2 L + ST)$ for class II algorithms. Optimisations have to be performed over $L = B$.

Table IV.5: Possible parallelisms for parallel merge.

Strategy (W)	Parallelisms over	Parallelisms and costs			$\mathcal{O}(\bullet)$ merge performance, $C=\infty$		Q
		Z_2	D_2	Z_3	ρ^P	χ	
For all HMM algorithms with $L_b=1$							
MP0	t	$nL\times S^2$	L	$S\log L$	$n\log L\times S^3$	$L/\log L$	L/M
MP0.S	t, s_t	$nL\times S^2$	LS	$S\log L$	$n\log L\times S^2$	$LS/\log L$	L/M
MP0.R	t, sub-runs r	$nL\times S^2$	LS	$S\log L$	$n\log L\times S^2$	$LS/\log L$	L/M
MP0.Q	t, sub-runs q	$nL\times S^2$	LS	$S\log L\log S$	$n\log L\times S^2\log S$	$LS/(\log L\log S)$	H
MP0.SR	t, s_t, r	$nL\times S^2$	LS^2	$S\log L$	$n\log L\times S$	$LS^2/\log L$	M
MP0.SRQ	t, s_t, r, q	$nL\times S^2$	LS^3	$S\log L\log S$	$n\log L\times\log S$	$LS^3/(\log L\log S)$	H
For class I algorithms with $L_b>1$							
MP1	b	$nB\times S^2$	B	$S\log B$	$n\log B\times S^3$	$B/\log B$	L/M
MP1.S	b, s_t	$nB\times S^2$	BS	$S\log B$	$n\log B\times S^2$	$BS/\log B$	L/M
MP1.R	b, sub-runs r	$nB\times S^2$	BS	$S\log B$	$n\log B\times S^2$	$BS/\log B$	L/M
MP1.Q	b, sub-runs q	$nB\times S^2$	BS	$S\log B\log S$	$n\log B\times S^2\log S$	$BS/(\log B\log S)$	H
MP1.SR	b, s_t, r	$nB\times S^2$	BS^2	$S\log B$	$n\log B\times S$	$BS^2/\log B$	M
MP1.SRQ	b, s_t, r, q	$nB\times S^2$	BS^3	$S\log B\log S$	$n\log B\times\log S$	$BS^3/(\log B\log S)$	H
For class II algorithms with $L_b>1$							
MP2	b	$nB\times L_bS^2$	B	$S\log B$	$n\log B\times L_bS^3$	$B/\log B$	L
MP2.S	b, s_t	$nB\times L_bS^2$	BS	$S\log B$	$n\log B\times L_bS^2$	$SB/\log B$	L/M
MP2.R	b, sub-runs r	$nB\times L_bS^2$	BS	$S\log B$	$n\log B\times L_bS^2$	$SB/\log B$	L/M
MP2.T	b, block length t	$nB\times L_bS^2$	$BL_b=L$	$S\log B$	$n\log B\times S^3$	$T/\log B$	L/M
MP2.Q	b, sub-runs q	$nB\times L_bS^2$	BS	$S\log B\log S$	$n\log B\times L_bS^2\log S$	$SB/(\log B\log S)$	H
MP2.SR	b, s_t, r	$nB\times L_bS^2$	BS^2	$S\log B$	$n\log B\times L_bS$	$S^2B/\log B$	L/M
MP2.ST	b, s_t, t	$nB\times L_bS^2$	$BL_bS=LS$	$S\log B$	$n\log B\times S^2$	$TS/\log B$	L/M
MP2.STR	b, s_t, t, r	$nB\times L_bS^2$	$BL_bS^2=LS^2$	$S\log B$	$n\log B\times S$	$TS^2/\log B$	M
MP2.STRQ	b, s_t, t, r, q	$nB\times L_bS^2$	$BL_bS^3=LS^3$	$S\log B\log S$	$n\log B\times\log S$	$TS^3/(\log B\log S)$	H

The theoretical efficiency limit for parallel merge is $\xi=1/Z_3$ for all strategies and is omitted from the table.

The formulas in the table present theoretical optima. In practice, P_2 and $\varrho^{(merge)}$ will be used in stead of D_2 and $\rho^{(merge)}$.

Strategies MP0.Q, MP1.Q and MP2.Q (all grey) should be avoided on their own due to their low efficiency and high communication costs, but may be applied together with other parallelisms (e.g. MP0.SRQ, MP01.SRQ and MP2.STRQ) when C is sufficiently large.

Parallel merge assuming $\mathbf{L_b > 1}$ for class I algorithms

In the $L_b > 1$ case for class I algorithms, the (IV.48) calculations have to be performed only for the block-ends, achieving an $\mathcal{O}(L/B)$ speedup and space requirement reduction compared to the $L_b = 1$ case, as in the case of sequential merge. The codename for the basic version (with no further parallelism beyond the tree-reduction over blocks) is MP1.

While the mechanics are different, the applicable parallelisms, their communication costs and their relative performance are similar to those for the $L_b = 1$ case. The notable differences are that recursions with $L_b > 1$ are required, there is an $\mathcal{O}(L/B)$ speedup, the space requirement is $\mathcal{O}(S^2 B)$ and optimisations are performed over B (again, $L = T$ is assumed optimal theoretically). See Table IV.5 for strategies, codes and details.

Parallel merge assuming $L_b > 1$ for class II algorithms

As for sequential merge, the case $L_b > 1$ for class II algorithms requires the merge step to be performed for every t in every block and introduces an additional possible parallelism over the block-length. The basic version (with no further parallelism beyond the tree-reduction over blocks) is MP2. The strategy including additional parallelism over present states is MP2.S, over sub-runs are MP2.R and MP2.Q, over block length is MP2.T, and the most noteworthy combinations are MP2.SR, MP2.ST, MP2.STR and MP2.STRQ. Parallelism over q is only advised when all other alternatives are exhausted, the highest theoretically achievable runtime is $\mathcal{O}(n \log B \times \log S)$, but is not expected to be achievable in practice with current technology for any interesting-sized HMMs.

The space requirements are $\mathcal{O}(LS^2)$ for both the recursion and merge steps and $\mathcal{O}(LS^2 + ST)$ overall, for all strategies. Strategies have to be optimised over both L and B simultaneously.

4.2.7 The final set of basic block-parallel strategies

The final set of basic HMM parallelisation strategies that include block-parallelism can be found by combining compatible recursion and merge steps – R0 recursion variants with MS0 and MP0, R1 with MS1 and MP1 for class I algorithms and R1 with MS2 and MP2 for class II algorithms.

Counting all possible combinations, there are 48 strategies using $L_b = 1$ fixed, there are 96 strategies with $L_b > 1$ for class I algorithms and 144 strategies with $L_b = 1$ for class II algorithms. Some of these strategies are trivially sub-optimal for every combination of S, T and C. For instance, due to the relative inefficiency and the high Q requirement of parallel summations, the recursions and merge steps should not be parallelised over summations,

unless all other options are exhausted, $Q^{(platform)} = H$ and there is still sufficient C left after applying all other parallelisms. When discounting sub-optimal strategies, there are still 32, 32 and 52 viable combinations using $L_b = 1$ fixed, with $L_b > 1$ for class I algorithms and with $L_b = 1$ for class II algorithms. Altogether, there are 64 viable options for class I and 88 for class II algorithms. Some of these strategy options may not offer parallel acceleration in theory (e.g. ones including the merge steps MS0 and MS2), but may allow for important optimisations in specific cases in practice.

With an arsenal of strategies at hand, the natural question is which of these strategies should we use in practice on a range of target platforms for a particular application. It is clear that answering this question in general for all possible problem size and platform combinations is rather complex even in theory – it involves constrained optimisations over B and/or L for all combinations of S, T, C and Q (theoretical performance analyses are presented in Section 4.3 for a number of chosen SCM platform models).

There is no doubt that further strategies can be designed in addition to the basic strategies described here and better performance can be achieved in specific cases, but the above lists should be a good starting point for developing alternatives (see Section 5.4 for future work for checkpointing).

4.3 The performance landscape of HMM parallelisation strategies

This section explores the performance landscape of different HMM parallelisation strategies, estimating and comparing their highest achievable theoretical acceleration under a number of model constraints.

4.3.1 The performance analysis setup

Platform models

The performance of parallel algorithms depends on a multitude of factors in practice, including the parallelisms exploited (i.e. the parallelisation strategy), the implementation details and the hardware and software platforms available. The performance analysis framework applied in this section was designed to allow focus on the theoretical performance differences between parallel strategies, with minimal assumptions about computation platforms and without regard to implementation details.

Performance was assessed for each strategy on a number of platform models in the SCM

framework (see in Section 4.1.1), with each platform modelled as a 2-tuple $(C,\ Q)$[11]. All further platform details were disregarded (including thread hierarchies, scheduling, memory types, etc.) and in order to compare highest achievable performance, unlimited memory was further assumed.

Table IV.6 presents the platform models used in this section, with corresponding Q and C limits. These models are chosen to cover the wide scale of present-day and expected near-future platforms but are not assumed to be correctly representing any specific existing system.

Table IV.6: Models of parallel platforms used for performance assessment.

Model platform	C	$Q^{(platform)}$	Inspiration
#1	32	M	A single CPU with 32 cores
#2	1,728	0	A small, CPU-based distributed cluster
#3	299,008	0	A large distributed system on the scale of Titan[†]
#4	1,728	L	A small, CPU-based cluster with medium comm. costs
#5	299,008	L	A large, CPU-based system with medium comm. costs
#6	1,728	M	A small, CPU-based cluster with low communication costs
#7	299,008	M	A large, CPU-based system with low communication costs
#8	2,688	H	A single GPU
#9	999,936	H	A medium-scale GPU cluster with low communication costs
#10	50,233,344	H	A large-scale GPU cluster with low communication costs

[†] The Titan supercomputer has 18,688 nodes with 16-core CPUs, which equals to 299,008 CPU cores.

In order to perform our analysis, it was necessary to fix the $Q = L/M$ communication costs for all strategies. The communication requirements used in this section for the various alternatives of recursion and merge steps are given in Table IV.7, where the L and M classifications

[11] As a reminder, C is the quantitative limit for effective parallelism ($C \in \mathbb{N}$) and Q is a qualitative measure of the level of inter-thread communication 'allowed' on the platform ($Q \in \{0, L, M, H\}$ as 'zero', 'low', 'medium' and 'high', with $0 < L < M < H$).

of communication strategies were roughly determined based on expected synchronisation time intervals. It was assumed that Q=M when the time interval between synchronisations is not greater than S^2 (and $S \geq 6$ was enforced for platforms with Q=L). Based on the Q of the recursion and merge steps, the overall communication requirements of a strategy (consisting only of a recursion step for S.S and S.SS' but consisting of both a recursion and a merge step for all other strategies) are given as

$$Q^{(W)} = \begin{cases} Q^{(recursion)} & \text{when } W \in \{ S.S,\ S.SS' \} \\ \max\left(Q^{(recursion)}, Q^{(merge)}\right) & \text{else.} \end{cases} \tag{IV.50}$$

Recall, that in SCM it is further assumed that a given strategy W is not applicable on a platform (implying $\varrho^{(W)} = \infty$) when $Q^{(platform)} < Q^{(W)}$, while communication costs are negligible and do not affect execution time when $Q^{(platform)} \geq Q^{(W)}$.

The SCM framework together the assumptions on strategies may be rather crude, but allows us to avoid many degrees of complexity when estimating algorithm run times[12]. It also allows us to avoid making detailed assumptions about computational platforms such as the time required for different memory operations, different levels of synchronisations and scheduling mechanisms, which could otherwise lead to unnecessary loss of generality.

When assessing the results on systems with large C, it is good to keep in mind that it is very unlikely that systems of very large scale will have low communication costs between all compute units and threads in the close future. It is more likely that there will be sub-units with low internal communication costs (e.g. GPUs, CPUs or hybrids) but the communication between these sub-units will likely not be negligible, unless strategies are designed to take these multi-layered structures into account. One option to exploit such architectures is to perform independent analyses on different nodes (e.g. over observations and/or parameterisations) and choose a strategy for performing computation on the sub-units according to their size, which is likely going to fall between small and medium scale platform models with a given Q.

Regarding models #8-10, it may be worth noting a couple of further specific limitations. Although it is true that real many-core systems may have a high number of compute cores and allow for very high levels of parallelism, the number of these cores usually do not translate directly into acceleration as it might be expected on small multi-core systems. It is also true, that many-core systems tend to facilitate efficient inter-thread communication, but that does not mean in most cases that communication costs are negligible and so communication costs may affect performance in practice to varying degrees. Further, current many-core systems tend to be more constrained in memory space than multi-core architectures, which may also

[12]Such as modelling execution as concurrent memory and calculation operations

Table IV.7: Inter-thread communication requirements, theoretical parallelisms and execution times of different parallel recursion and merge steps.

Alternatives for the recursion step	$t^{(sync)}$	$Q^{(rec)}$	$\rho^{(rec)}$	$D^{(rec)}$
S.S	S	M	S^2T/D	S
S.SS'	1	H	S^2T/D	S^2
R0	S^2	0	S^2T/D	L
R0.R/S	S	0	S^2T/D	LS
R0.RS	1	0	S^2T/D	LS^2
R1	S^3	0	S^3T/D	B
R1.R	S^2	0	S^3T/D	BS
R1.RS	S	M	S^3T/D	BS^2
R1.RSS'	1	H	$S^3T\log S/D$	BS^3
Alternatives for merge	$t^{(sync)}$	$Q^{(merge)}$	$\rho^{(merge)}$	$D^{(merge)}$
MS0	S^2	0	S^2T/D	1
MS0.S	S	M	S^2T/D	S
MS0.SR	1	H	$S^2T\log S/D$	S^2
MS1	S^2	0	$TBS^2/(LD)$	1
MS1.S	S	M	$TBS^2/(LD)$	S
MS1.SR	1	H	$TBS^2\log S/(LD)$	S^2
MS2	L_bS^2	0	S^2T/D	1
MS2.S	L_bS	L	S^2T/D	S
MS2.T	S^2	M	S^2T/D	L_b
MS2.SR	L_b	H	$S^2T\log S/D$	S^2
MS2.ST	S	M	S^2T/D	SL_b
MS2.STR	1	H	$S^2T\log S/D$	S^2L_b
MP0	S^3	L	$S^3T\log L/D$	L
MP0.S/R	S^2	M	$S^3T\log L/D$	LS
MP0.SR	S	M	$S^3T\log L/D$	LS^2
MP0.SRQ	1	H	$S^3T\log L\log S/D$	LS^3
MP1	S^3	L	$TBS^3\log B/D$	B
MP1.S/R	S^2	M	$TBS^3\log B/D$	BS
MP1.SR	S	M	$TBS^3\log B/D$	BS^2
MP1.SRQ	1	H	$TBS^3\log B\log S/D$	BS^3
MP2	L_bS^3	L	$S^3T\log B/D$	B
MP2.S/R	L_bS^2	L	$S^3T\log B/D$	BS
MP2.T	S^3	L	$S^3T\log B/D$	$BL_b = L$
MP2.SR	L_bS	M	$S^3T\log B/D$	BS^2
MP2.ST	S^2	M	$S^3T\log B/D$	LS
MP2.STR	S	M	$S^3T\log B/D$	LS^2
MP2.STRQ	1	H	$S^3T\log B\log S/D$	LS^3

$t^{(sync)}$ is the time interval between inter-thread synchronisations.

limit their performance in certain cases and especially for memory-intensive applications, which is the category into which HMMs are expected to fall.

Despite the obvious limitations of this theoretical framework, it allows for relatively detailed theoretical analysis for a wide scale of platforms and problem sizes.

Assessing performance

Since the $\varrho^{(W)}(S,T)$ are constrained both by C and the assumptions $L,B\in\mathbb{N}$ and $L,B\geq 2$, finding θ^* is rather difficult in practice in many cases. For the sake of our performance explorations, L^* and B^* were found by brute-force search, for every combination of S and T. For the sake of tractability, the (L,B) parameter space was constrained to $L\in\{2,3,...,\lfloor\sqrt{T}\rfloor-1,\lfloor\sqrt{T}\rfloor,[1/\lfloor\sqrt{T}\rfloor],[1/(\lfloor\sqrt{T}\rfloor-1)],...,[T/3],[T/2],T\}$ and $B\in\{2,3,...,\lfloor\sqrt{L}\rfloor-1,\lfloor\sqrt{L}\rfloor,[1/\lfloor\sqrt{L}\rfloor],[1/(\lfloor\sqrt{L}\rfloor-1)],...,[L/3],[L/2],L\}$, which is roughly synonymous to assuming $n=T/L\in\mathbb{N}$ and $L_b=L/B\in\mathbb{N}$ and should not constrain the validity of our assessments.

The actual values of χ^* presented in this section are estimations of theoretical upper limits and are not supposed to imply actual practical performance.

4.3.2 Theoretical performance comparisons between parallel strategies

The performance landscape of parallel strategies was explored by comparing χ^* parallel acceleration surfaces in (S,T) space. The strategies were ordered roughly according to complexity[13] so that where the χ^* surfaces of different strategies overlap, the lower-numbered strategies are visible.

It is important to remember that strategies assuming $L_b=1$ (strategies 1-23) are applicable and identical for class I and class II algorithms, but others (strategies 24-51 for class I and 24-75 for class II) may work quite differently (see Sections 3 and 4.2 on HMM parallelisation strategies for more details on the two classes and the corresponding strategies).

Analysis with no platform constraints

The first setting in which the performance of parallel HMM strategies is explored assumed $C=\infty$ and $Q=\infty$. Figure IV.6 shows the estimated optimal accelerations for different strategies at various configurations of S and T.

According to the relative position of the χ^* surfaces, the single optimal strategy appears to be

[13] See the legend of Figure IV.6 for the ordering and colour coding of strategies for class I and class II algorithms.

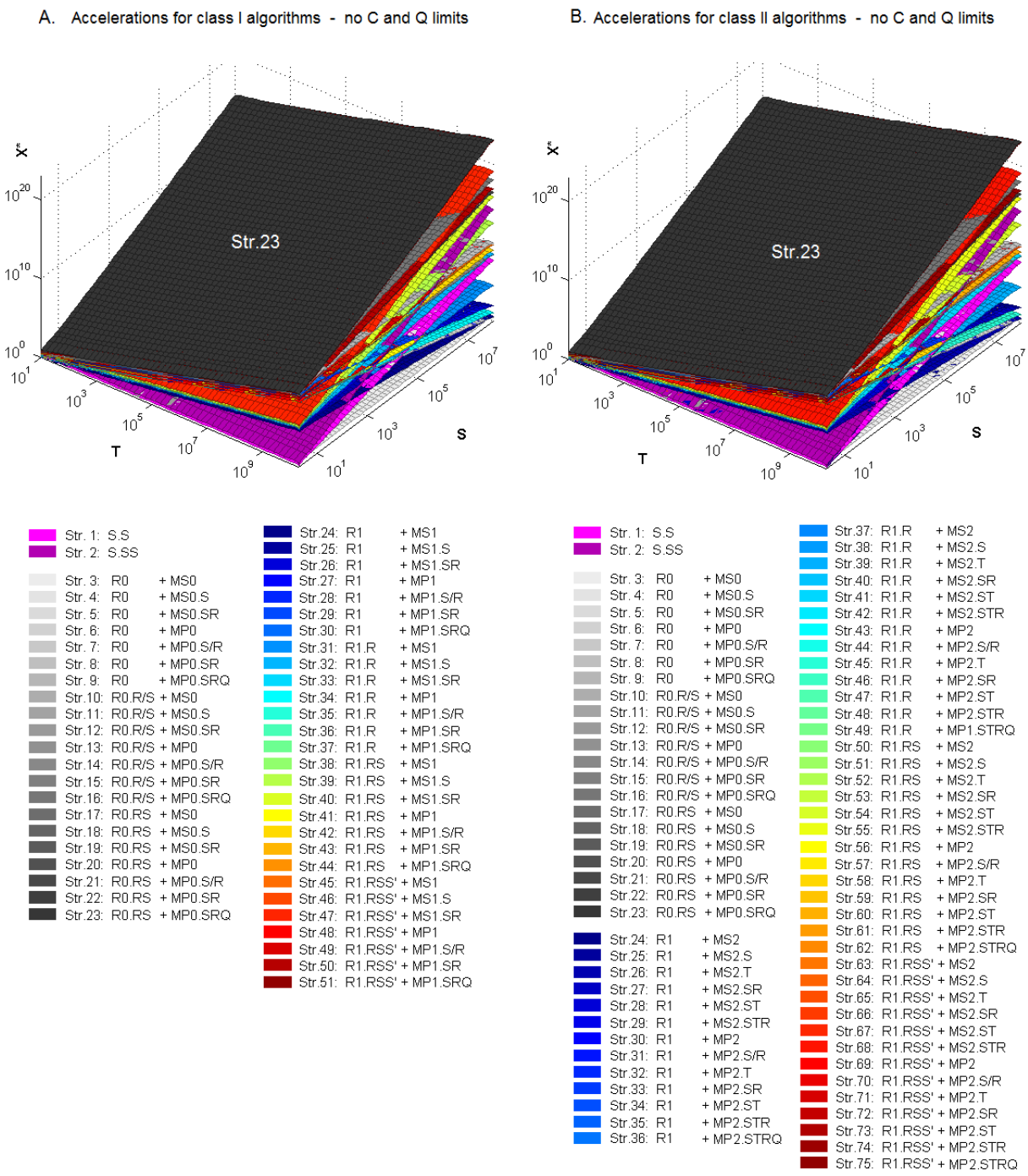

Figure IV.6: Acceleration comparisons of HMM parallelisation strategies in the theoretical setting where unlimited parallelism and negligible communication costs are assumed. **(A)** corresponds to strategies for class I algorithms and **(B)** corresponds to class II algorithms. The legends show the assignments between colours, strategy numbers and codes used for plots in this section.

strategy 23 for all problem sizes. This strategy is R0.RS+MP0.SRQ, where $L_b = 1$, merge is performed with parallel reduction and both the recursion and merge steps are all maximally parallelised. While strategy 23 appears optimal for both class I and class II algorithms, a handful of alternative strategies appear very close in performance. For class I, strategies 9,

16, 37 and 44 are close to optimal when S is small and T is large, and strategy 51 everywhere except where T is very small. For class II, strategies 9, 16, 49 and 62 are close to optimal when S is small and T is large and strategy 75 is close to optimal everywhere except where T is very small (supporting results are not presented here).

The optimality of strategy 23 is expected for two reasons. First, R0.RS can have the lowest ρ from all recursion steps, and second, with no C and Q limits, maximum parallelisation is expected to be the optimal choice.

Large-scale evaluations with platform constraints

In the second stage of performance exploration experiments, the platform models from Table IV.6 were used for specifying C and Q constraints.

The first obvious observation is that the χ^* surfaces of Figures IV.7-IV.10 and IV.12 show significant differences in the optimality landscape between parallelisation strategies and especially when compared to the unconstrained experiment (Figure IV.6).

Small single-unit multi-core systems

In the case of platform model #1 (the small single-unit model, Figure IV.7), strategies with frequent synchronisations are not allowed. The first observation is that there is an apparent S threshold (which will be referred to as the block-parallelism 'cutoff point') below which it is beneficial to use block-parallelism and above which a S.S and S.SS' are expected to perform better. Naturally, the region around the cutoff point can be seen as a contested territory, where optimality between BP and non-BP approaches may be more dependent on implementation, platform and other details. It is expected that the simpler approaches (S.S, S.SS') may be easier to optimise and, hence, may be expected to perform better in these regions.

The second observation is that while the Q constraint eliminated strategy 23 from this experiment, a number of strategies with $L_b{>}1$ show significantly better performance than strategy 22, which is the most highly parallelised $L_b{=}1$ algorithm allowed. It is easy to see that it is the C constraint that limits the performance of $L_b{=}1$ algorithms to the point that alternatives are better.

For class I algorithms, strategies 40 and 43 (R1.RS+MS1.RS and R1.RS+MP1.SR, the highest parallelised strategies with sequential and parallel merge) perform the best for all T and S (below the cutoff). Strategies 26 (R1+MS1.SR), 32 (R1.R+MS1.R), 33 (R1.R+MS1.SR), 35 (R1.R+MP1.R), 36 (R1.R+MP1.SR), 38 (R1.RS+MS1), 39 (R1.RS+MS1.R), 41 (R1.RS+MP1)

and 42 (R1.RS+MP1.R) are also roughly equivalent with the theoretical optimum when T>1,000 and 24 (R1+MS1), 25 (R1+MS1.R), 27 (R1+MP1), 28 (R1+MP1.R), 29 (R1+MP1.SR), 31 (R1.R+MS1) and 34 (R1.R+MP1) when T>10,000.

For class II algorithms, strategies with sequential merge have significantly better theoretical performance than those with parallel merge. The optimal strategy is 54 (R1.R+MS2.ST, the most highly parallelised variant of the MS2 strategies) for almost all T, but a number of simpler strategies, such as 26 (R1+MS2.T), 28 (R1+MS2.ST), 39 (R1.R+MS2.T), 41 (R1.R+MS2.ST) and 52 (R1.RS+MS2.T) may perform equally well when T>1,000.

It is safe to assume that in practice, block-parallelism on small multi-core systems is justifiable only for time-critical applications where even small acceleration differences matter, and only when S is extremely small (e.g. $S \leq 4$ on current platforms) and implementations are very efficient. In such cases, highly parallelised strategies should be used if possible (e.g. 40: R1.RS+MS1.RS and 43: R1.RS+MP1.SR for class I and 54: R1.RS+MS2.ST for class II), but simpler strategies (e.g. 26: R1+MS1.SR for class I and 26: R1+MS2.T for class II) may also be used when T>1000.

Distributed and multi-unit multi-core systems

Models #2-7 represent multi-node structures with different levels of connectivity between nodes and, consequently, different values of Q. For simplicity, it is assumed that there is only single level and single type of communication between all compute elements in every model. Models #2-3 can be imagined as systems that include a large number of remote, distributed nodes, each capable of single-thread computation only. Models #4-7 assume that infrequent communications between threads are allowed and can be performed efficiently. Models #4-5 may resemble classic multi-node compute servers with $Q = L$ while #6-7 assume very closely connected nodes (such that communications are roughly as fast between nodes as between the cores of a CPU). These models are expected to be sufficient to explore parallelism options for most existing and future transistor-based multi-node platforms (possible future technologies such as quantum computing may offer different interesting alternatives).

Figure IV.8 demonstrates how the performance landscape changes with the platforms' increasing Q limits for class I algorithms. It is clear from the Q requirements (Table IV.7) that block-parallelism is the only option for parallelising HMM algorithms when inter-thread communication is costly on a platform. Although the actual χ^* values are not expected to be representative, the experimental results indicate that block-parallelism may allow for significant accelerations on these systems.

Strategy 31 (R1.R+MS1) appears to be close to optimal for all problem sizes for both Q=0 and Q=L and strategy 24 (R1+MS1, the simplest block-parallel strategy with L_b>1) is close

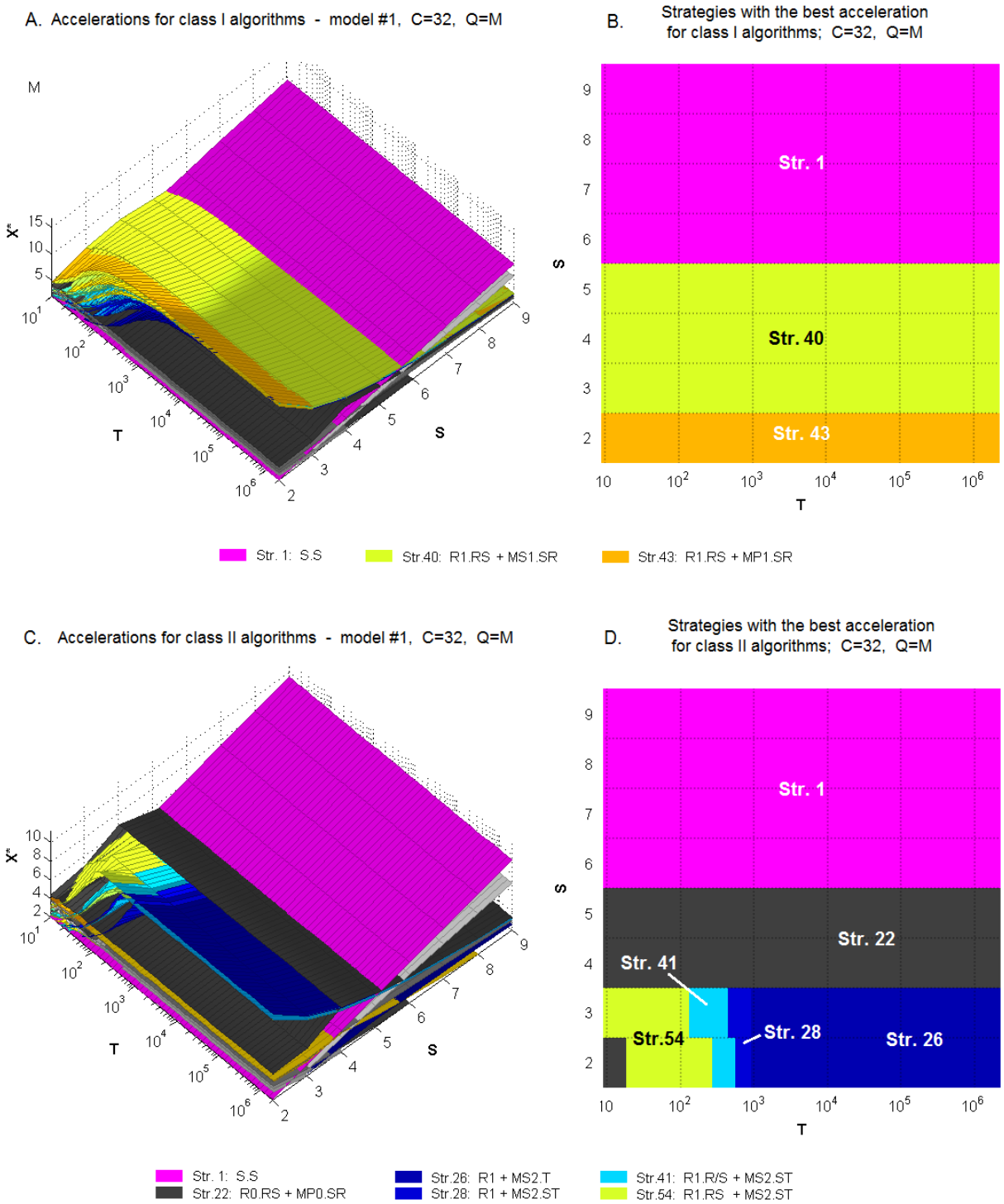

Figure IV.7: Parallel HMM accelerations on a single processing unit with 32 cores and medium level (Q=M) inter-thread communications and synchronisations allowed (platform model #1). **(A)** and **(C)** present $\chi*$ surfaces for class I and II algorithms, respectively, while **(B)** and **(D)** present corresponding optimality maps. For simplicity, the legends only include the most relevant algorithms – for the complete list, see Figure IV.6.

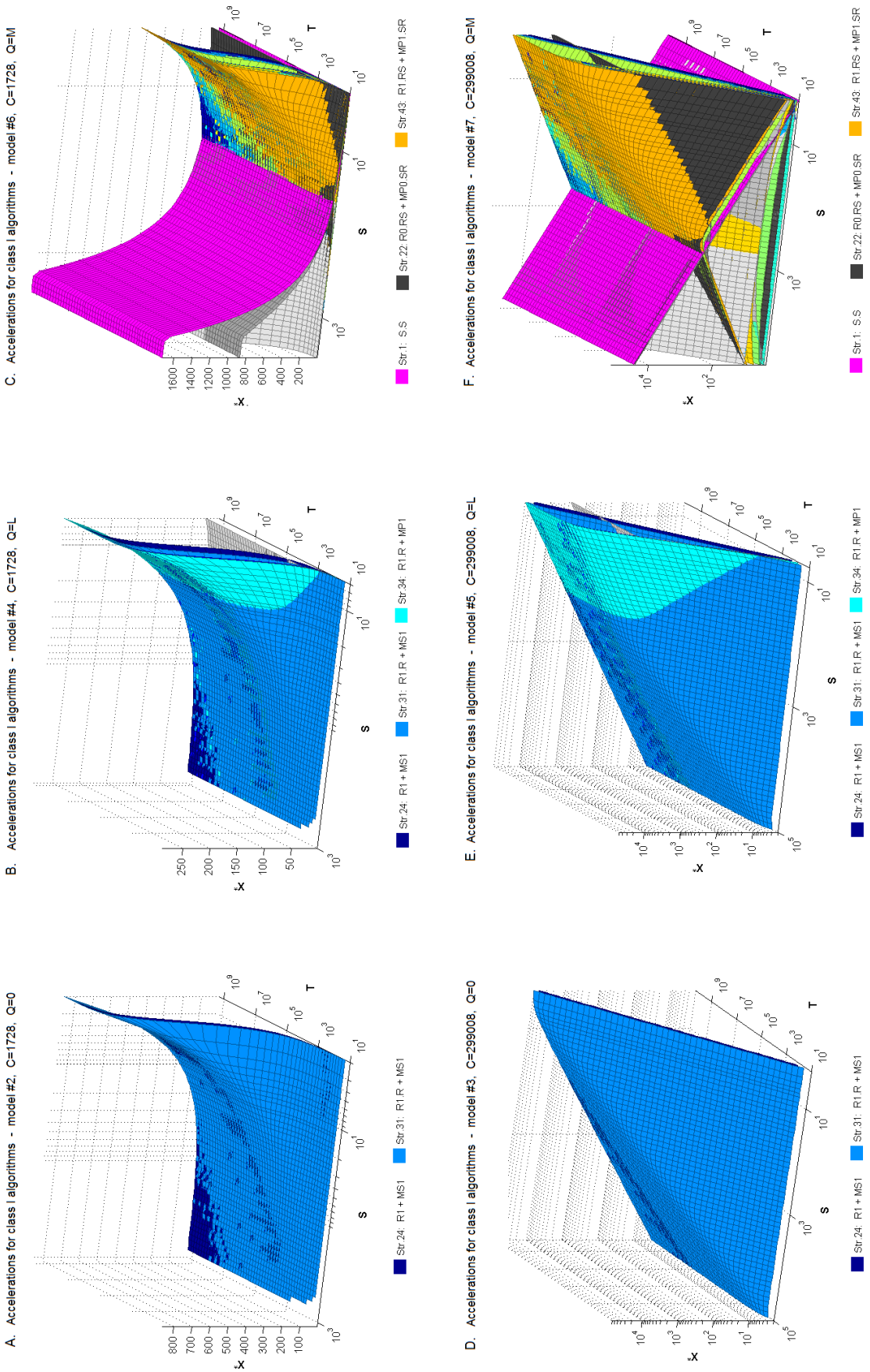

Figure IV.8: Parallel HMM accelerations on distributed and multi-core systems with various strategies for class I algorithms. **(A-C)** present experimental results for models #2, #4 and #6 with C=1,728 and **(D-F)** present experimental results for models #3, #5 and #7 with C=299,008. For simplicity, the legends only include the most relevant algorithms – for the complete list, see Figure IV.6.

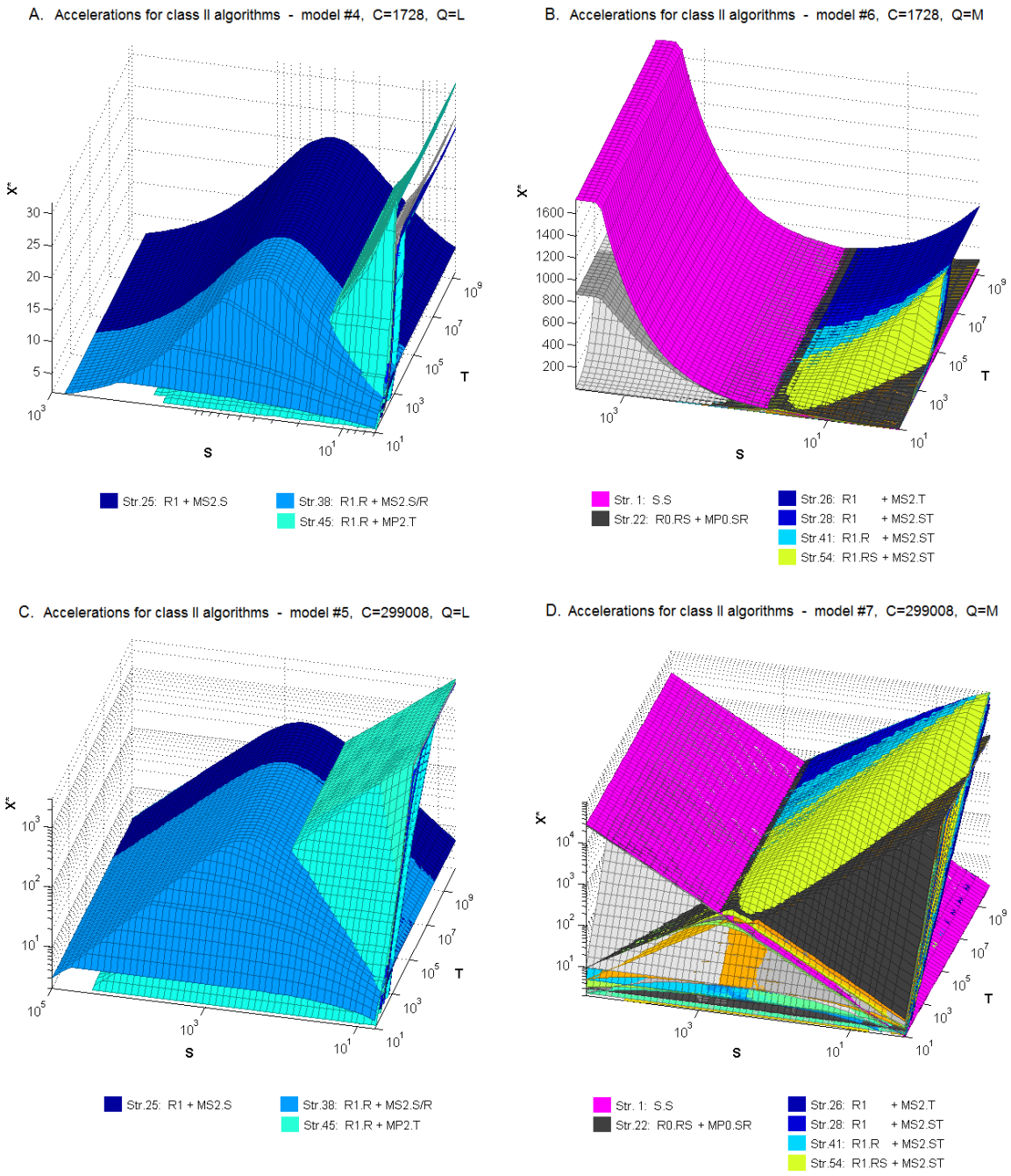

Figure IV.9: Parallel HMM accelerations on distributed and multi-core systems with various strategies for class II problems. **(A-B)** present experimental results for models #4 and #6 with C=1,728 and **(C-D)** present experimental results for models #5 and #7 with C=299,008. For simplicity, the legends only include the most relevant algorithms – for the complete list, see Figure IV.6.

to optimal when S and T are both large.

For Q=L, strategy 34 (R1.R+MP1) performs slightly better than others when S is small and this strategy is close to optimal for all S when T is large. Finally, strategy 27 (R1+MP1) is also close to optimal when Q=L and T is large.

When Q=M is allowed, the landscape is very similar to the one observed for a small single-core system. The differences are that the cutoff point is much higher (depending on C) and that strategy 43 is dominant for most ranges of T below the cutoff point. The most notable exception is at very small data sizes, where 22 offers a slight improvement. When T is large enough, strategies 24-29, 31, 32, 34-36, 38-39, 41 and 42 are close to or identical to the optimum.

It is good to keep in mind that when $L^* = B^*$ are very small (when T is small), the communication costs of R1.R, MS2.S, MS2.R and MP2.T may become significant in practice, with R1+MS1 remaining as the only option.

In the case of class II algorithms, no strategies with $Q = 0$ provide significant acceleration for any combination of S and T (compared to sequential execution). The reason for this is that the only two merge options allowed for $Q = 0$ are MS0 and MS2, which both have $\mathcal{O}(TS^2) = \rho^{(sequential)}$ run time.

Figure IV.9 presents the performance landscape for $Q \in \{L, M\}$. For Q=L, strategy 45 (R1.R+MP2.T) is optimal when S is small and T isn't too small and strategy 44 (R1.R+MS2.S) behaves almost identically in performance. Everywhere else, strategy 38 appears optimal, with strategy 25 (R1+MS2.S) being close or equivalent when T is large. On really large systems, strategies 6, 13, 20, 30, 31 32 and 43 also approach the optimum.

For Q=M, the landscape is, again, very similar to the small single-unit model. Block-parallel algorithms offer a significant improvement compared to state-wise parallelism (S.S) below the cutoff point, the position of which is dependent on C. The same strategies with $L_b > 1$ (strategies 22, 26, 28, 41, and 54) prove to be optimal below the cutoff point, but 22 shows more and 26, 28 and 41 show less relevance when C is large. Strategies 15 and 61 are close to the optimum for very small data sizes and strategies 39, 52 when T is large; all other strategies remain sub-optimal for most data sizes.

Parallel systems with efficient inter-thread communication

Models #8-10 assume high C, practically no limit on inter-thread communications and are meant as highly idealised representations of current many-core systems of different scale. Due to the significant differences between the idealised platform models #8-9 and real multi-core systems, it is good to keep in mind that the results on these models are purely theoretical. Due to the absence of a Q limit, this scenario explores the optimality of all strategies presented in Section 4.2 under C constraints only. This scenario is identical to assuming a PRAM model with C-fold parallelism. Figures IV.10 and IV.12 present the performance landscapes for class I and class II algorithms for these models.

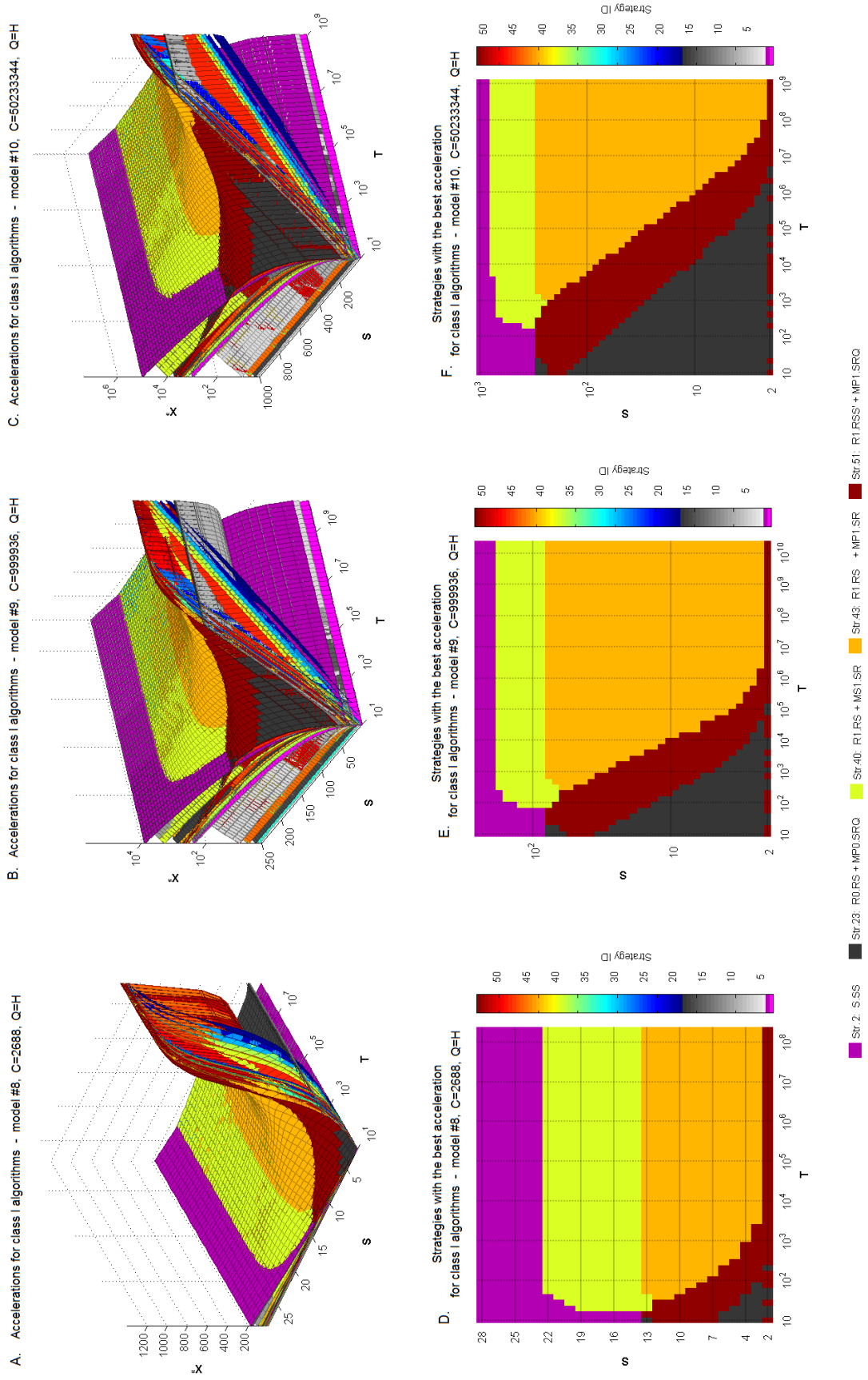

Figure IV.10: Parallel HMM accelerations for class I algorithms, including all strategies.

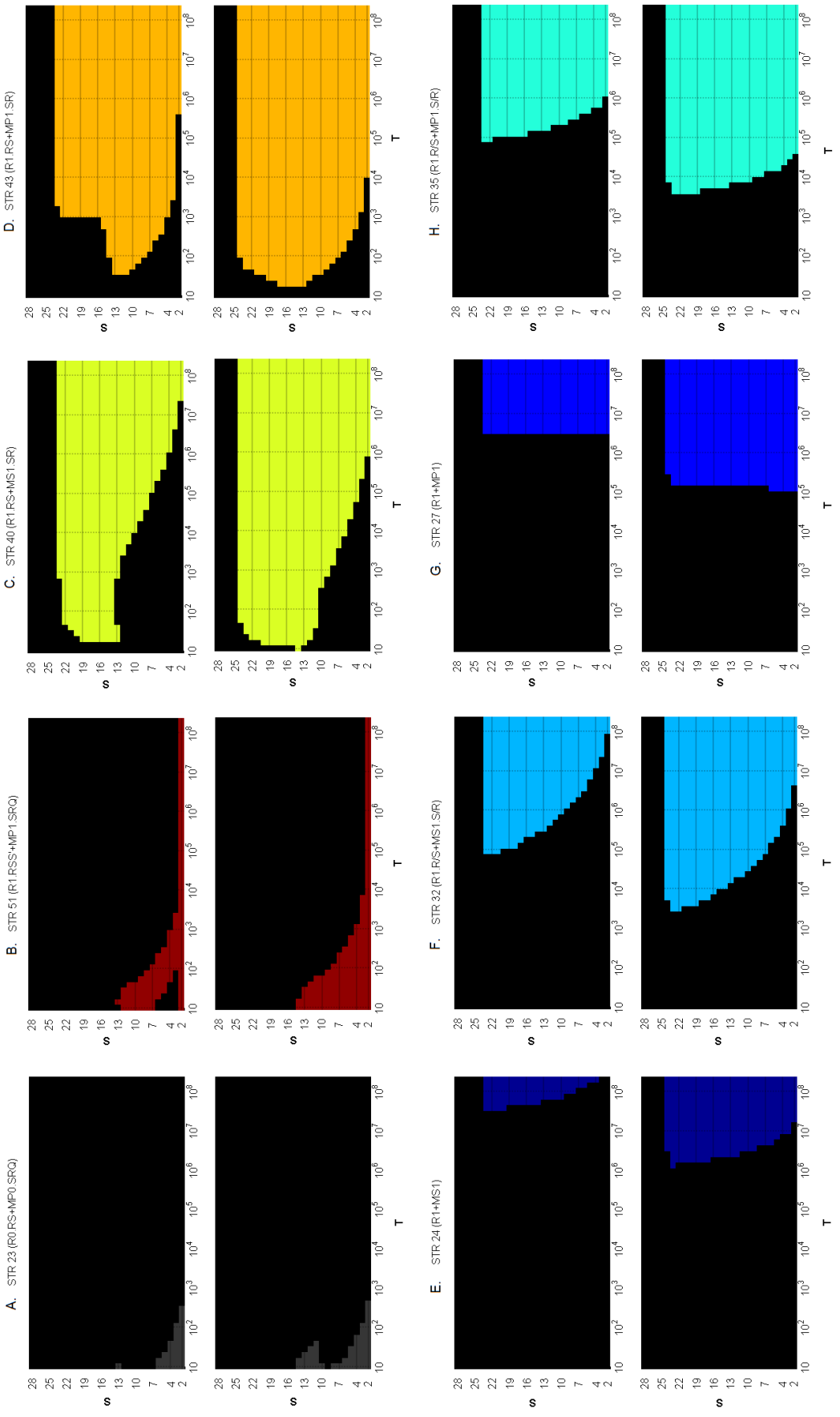

Figure IV.11: Regions of expected optimality for selected HMM parallelisation strategies for class II algorithms, including all strategies ($Q^{(platform)}$=H) and assuming C=2688. Every subpolot consists of two 'optimality maps' for a given strategy. The top ones highlight problem sizes (combinations of S and T) where the the theoretical acceleration limit is at least 99% of the optimal strategy (for that S and T), the bottom ones highlight areas where performance is greater than 80%.

For class I algorithms, the optimal strategies appear to be 23 and 51 when S and T are both small and strategy 51 also appears optimal for all T when S=2. Everywhere else below the cutoff point, strategies 40 and 43 appear optimal or close to optimal (see Figure IV.11 (**C-D**) for C=2688). Since strategy 23 is optimal only when both S and T are very small, its usability on small platforms (when C is small) may be very limited in practice, but it may be relevant for use on medium or large-scale multi-core platforms (Figure IV.10 (**B-C**)).

At higher values of T, many alternative block-parallel strategies with lower complexity become close to optimal. Strategies 42 and 44 have close to optimal performance from medium ranges (around $T > 10^4$ for C=2688), strategies 27-29, 33-37, 39 and 41 are close for higher ranges (around $T > 10^5$ for C=2688) and strategies 24, 25, 30-32 and 38 at very large values of T. The most noteworthy alternative strategies that provide sufficient alternatives when T is large may be 24, 27, 32 and 35. See Figure IV.11 **(E-H)** for the optimality of the best performing strategies with C=2688.

For class II algorithms, the optimality between block-parallel strategies is more complicated to determine below the S cutoff point (see Figure IV.12). More detailed analysis (see Figure IV.13) reveals that strategies 54 and 55 are close to optimal at most of the (S, T) space on small platforms, except when S and T are both very small. The smaller problem ranges are dominated by strategies 23 and 68, while 74 and 75 also appear optimal in small ranges. As expected, some simpler strategies are also close to optimal at large values of T, such as strategies 52, 41, 39, 28 and 26.

For class II, the optimality landscape is qualitatively different on larger (medium and large) platforms. Here, strategy 23 dominates a fair part of (S, T) space at the lower ranges, while the relevant ranges of other strategies are shifted towards larger values of T.

Strategies 54 and 55 are optimal throughout the higher ranges, with strategies 28, 29, 39, 41, 42 and 52 showing some relevance on medium-scale platforms, but not for large C. It is notable that parallel-merge strategies with L_b=1, such as strategies 6, 7, 8, 9, 13, 14, 15 and 16 are close to optimal at certain ranges, typically when S is small and T is large. It is possible that simpler strategies would be relevant at values of T that fell out of the scope of the analyses conducted, but those observation lengths are not expected to be relevant to most applications in the close future.

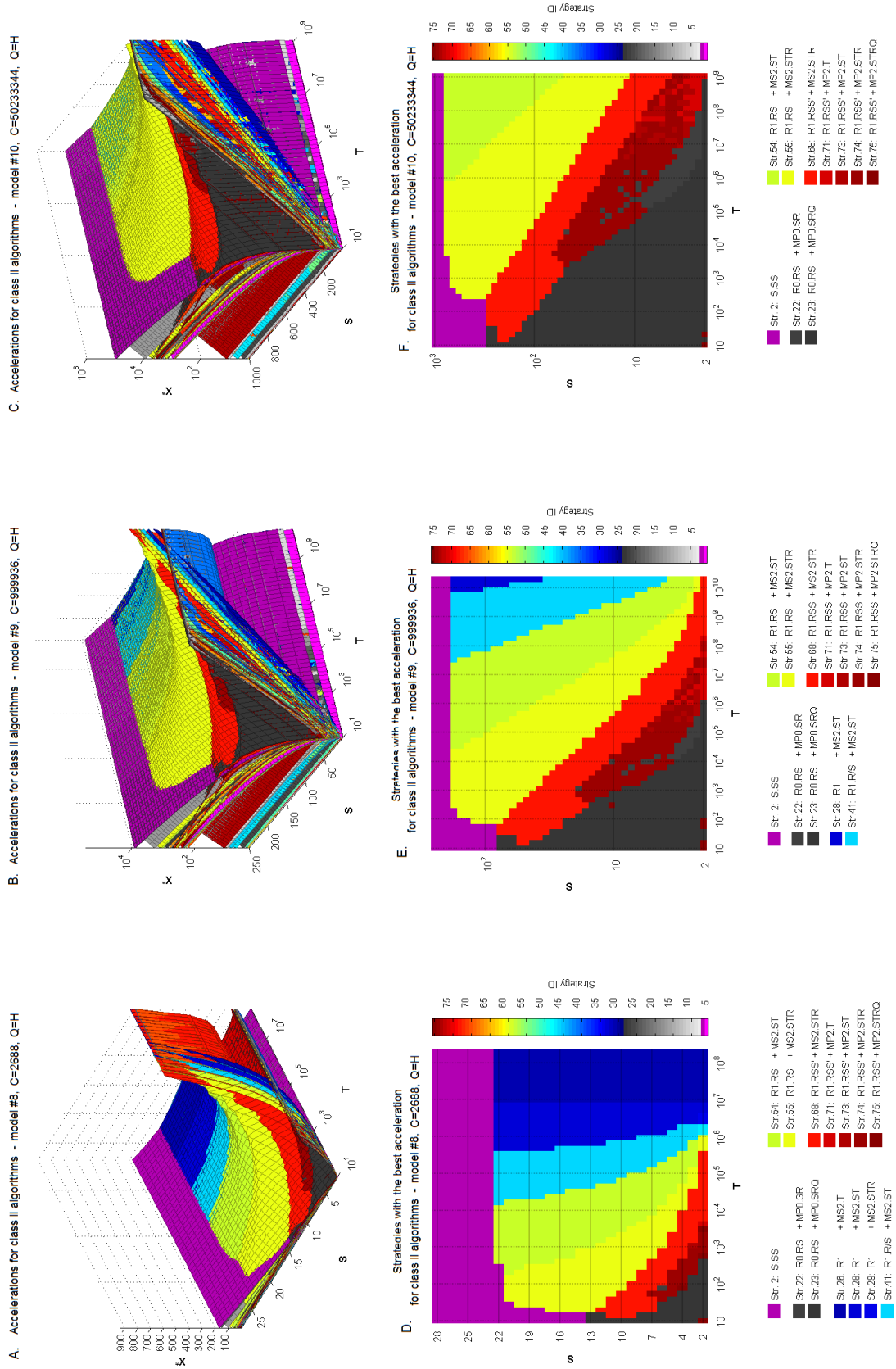

Figure IV.12: Parallel HMM accelerations for class I algorithms, including all strategies.

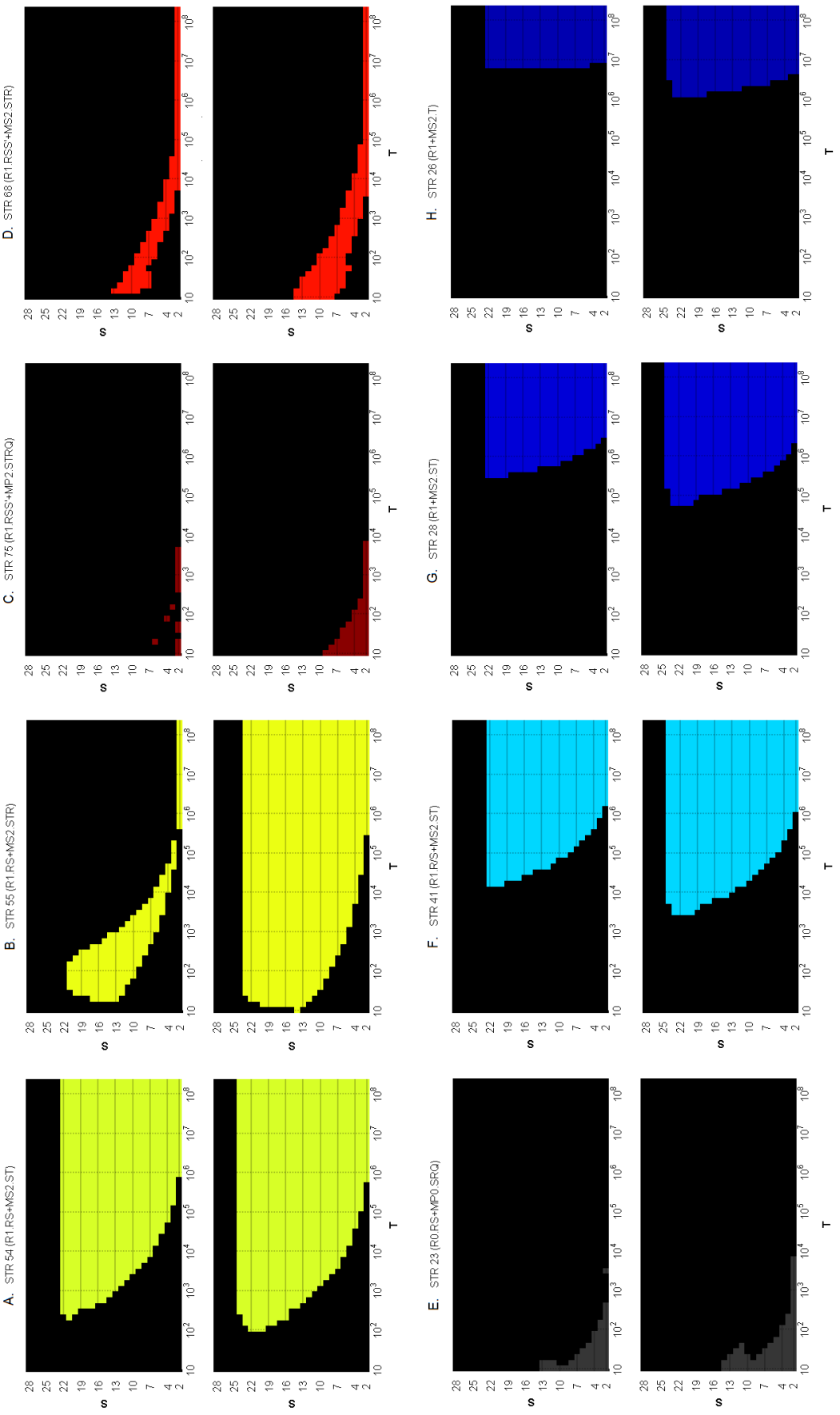

Figure IV.13: Regions of expected optimality for selected HMM parallelisation strategies for class II algorithms, including all strategies ($Q^{(platform)}$=H) and assuming C=2688. Every subpolot consists of two 'optimality maps' for a given strategy. The top ones highlight problem sizes (combinations of S and T) where the the theoretical acceleration limit is at least 99% of the optimal strategy (for that S and T), the bottom ones highlight areas where performance is greater than 80%.

5 Discussions and future work

5.1 Relevance and significance

It is clear from the results (Section 4.3.2) that block-parallelism is applicable and relevant for use on almost any parallel platform and may achieve significant performance improvements for all HMM algorithms. In certain cases, block-parallelism is also the only parallelisation option for achieving significant accelerations.

Applying block-parallelism is far from trivial, and may require more space than sequential algorithms or alternative parallelisms. When trivial parallelisms are applicable (over K observations and G parametrisations), those should be applied first (across the most remote nodes). When K and G are both low and execution time is important, further parallelisms need to be applied. One option is to use state-wise only parallelisms if inter-thread communication is efficient on target platforms and S is large. In all other cases, the optimal option is to apply block-parallelism. Block-parallelism is the only option when communication costs are large between parallel threads (as also mentioned in Nielsen and Sand (2011)). Block-parallelism is also relevant even for small multi-core systems such as a single CPU, but the relevance of different strategies greatly increases for large C.

Most HMM applications assume a moderate number of states (see Figure IV.1). Even in population genetics and speech recognition, where it may be possible and even seem sensible to use a greater number of states, a large S may lead to over-fitting in most applications. Consequently, it is reasonable to assume that the overwhelming majority of HMM applications will be in the $S \in \{2, ..., 1000\}$ and most likely in the $S \in \{2, ..., 100\}$ intervals.

The theoretical performance results show that block parallelism may be applicable for problems with a low number of states ($S < 20$) even on single present day GPGPUs (or moderate CPU clusters) and for problems with a medium number of states ($S < 100$) on moderate-sized GPU clusters. With the improvements of parallel architectures roughly following Moore's law (see Figure I.6), it is expected that improvements in the close future will make block-parallelism relevant for most applications on widely available compute platforms and block-parallelism may be a necessity for performance-critical applications.

It is clear that in the recursion steps all conditional sub-runs – corresponding to different blocks and/or starting values – are independent. Because there is no need for communication between threads of different sub-runs, nor between blocks until a merge step, some versions of block-parallelism may be suitable even for distributed computing (or "cloud computing") when using the right strategies, which may be an advantage over the alternative way of

achieving parallelism over S. When work is split across remote nodes or between nodes with inefficient inter-node communication, state-wise parallelisms are not expected to be applicable. When K and G are insufficient for utilising the available parallelisms, low-complexity block-parallel strategies may be the only option to achieve further accelerations.

On platforms with efficient communication channels, the most complex strategies tend to appear optimal for small problem sizes, but even very simple strategies become relevant at higher values of S and T. Due to the high expected practical communication costs and the possible complexity of implementations and optimisations, it is unlikely that the most complex strategies are going to be optimal in practice. However, medium complexity strategies are still expected to achieve significant accelerations and offer a definite improvement over state-wise parallelisms when S is not too large.

In some applications, block-parallelism may be trivialised by disregarding sequentiality between blocks. In such cases, HMM algorithms may be run in parallel on blocks, without having to perform any merge step other than the concatenation of block-wise results. This approach works best when independence and low correlation time points are identifiable ahead of calculations[14]. Alternatively, in applications where some degree of inaccuracies are acceptable, results may be improved by overlapping blocks and disregarding parts (e.g. half) of the overlapping regions from every block start and end.

With sufficient further development, block-parallelism may bring the power of the parallel-computing revolution to be widely applicable for HMMs, greatly improving the potential of their applicability and power for large problems.

5.2 General observations regarding optimality

The most obvious observation from the theoretical performance explorations is that on platforms where parallelisms over S are applicable, there is usually a cutoff value of S above which purely state-wise parallelism strategies are optimal.

The most important result may be that although strategies with L_b=1 appear optimal in theory, strategies with L_b>1 turn out to have better performance in most situations.

From the merge strategies, MS1 strategies appear to do better at medium ranges of S while MP1 strategies are best when S is small, as expected.

For class II algorithms, parallelism over block length is crucial for performance in the case of

[14]For instance, in the case of the Li and Stephens model, it is possible to choose the block boundaries to be at recombination hot-spots, which would reduce the sequential dependency between blocks.

the presented strategies. It may be unintuitive, but these strategies are still applicable and may still achieve significant accelerations on present day platforms.

5.3 Limitations

Limitations and applicability of basic block-parallelism

The four biggest limiting factors for the applicability of block-parallelism are its optimality at small ranges of S, the need for batch processing, the increased space requirement for class II algorithms and the potential complexity of implementations.

The theoretical performance results showed that block-parallelism is optimal only in small and moderate ranges of S. When S is too large, the simpler state-wise only parallelisms may perform better.

Block-parallelism requires batch-processing, i.e. that observations are available at least for all t in the current section (e.g. for all $t \in \{1, ..., L\}$). Hence, block parallelism is not directly suitable for on-line algorithms (see Section 5.4 for future work) and for applications requiring online processing, the algorithms will need to be modified if at all possible (see under 5.4).

Another limiting factor for the applicability of block-parallelism is the increased space requirement for class II algorithms (which include the forward-backward, Viterbi and Baum-Welch algorithms). The extra space requirement is not expected to be significant for very small S, but can be detrimental for larger problems. It may be possible to reduce space requirements by limiting L, but that may lead to reduced performance. Alternatively, some parallel-optimised version of checkpointing may be applied, which may reduce space requirements and may also improve execution time performance (see under 5.4).

The implementation of block-parallelism may be more complicated than the implementation of sequential algorithms and parallelisms over K, G and the state space. This is especially the case where strategies involve reductions, or applications involve modified versions of HMMs or HMM algorithms. In the absence of block-parallel HMM libraries, and when developing complicated applications with modified versions of HMM algorithms, one may use parallel linear algebra libraries (e.g. cuBLAS) to perform calculations according to the matrix-formulations of applicable strategies.

Limitations of optimality results

The most obvious limitation of the performance results is that they were produced in a theoretical setting. While the theoretical approach may be the most suitable one when aiming for wide generality, it can not be expected that all results will hold accurately in practice. The SCM framework disregards details of communication costs (communication costs will very rarely be negligible), platform architectures, streaming, memory requirements and many more. It is quite possible that strategies that appear identical in terms of performance in theory, will have greatly different accelerations in practice, and it is even possible that the order of optimality switches between strategies.

Disregarding space requirements, in particular, may be the strongest limitation of the results, but it is easy to see that an acceleration-centred performance analysis is the best starting point for exploring the applicability and power of block-parallelism.

The next important limitation may be that the Q communication requirements of strategies were determined intuitively based on rate of growth and not on actual estimates of the synchronisation frequencies. A more rigorous analysis may be performed with estimating synchronisation frequency for each $W\times S\times T$. The platform Q limits are also not expected to accurately represent existing parallel platforms. When the optimality of strategies for future applications is assessed for a specific platform, the exact C and the communication costs may be known and hence more detailed analyses may be performed under the BSP model, resulting in slightly changed χ^* surfaces in a greatly reduced S,T space, with expectedly different optimality results.

Many times there will be a bound on L due to memory constraints. These constraints have not been taken into account, although they may have significant impact in practice. This limitation does not invalidate the results in this chapter as the aim of this work was to explore and compare the theoretical upper performance bounds of block-parallel HMM algorithms. Nevertheless, it will be an important step in follow-up research to devise ways to maximise performance while minimising space requirements and also to compare parallel strategies under space limits (see Section 5.4 for future work).

The use of rate of growth formulas as the basis for acceleration estimations obviously means that constant factors are disregarded. This may have some validity implications for two reasons. First, more complicated strategies may result in more complicated code and higher constant factor multiple of operations and run time. Second, the rate of growth formulas are not expected to be representative for small values of S and T, hence results at the lower ranges should be observed more cautiously, as results in practice may be significantly different.

It was not enforced in the performance assessments that all $B, L_b, T, n \in \mathcal{Z}$. Although the χ^* results may not be exactly accurate as a consequence, the surface plots are smoother and present general trends more clearly. Further, for class II algorithms at large values of S and T, the L^* and B^* were calculated only for a subset of the possible values. Consequently, the surfaces above T=10,000 and S=1000 may be less accurate.

It is expected that practical implementation constraints such as the parallel technology of the target platforms greatly affect the efficiency of algorithms and the optimality of strategies. These performance implications may well lead to different answers in terms of what the optimal algorithm may be, often in favour of simpler algorithms.

5.4 Future Work

5.4.1 Improvements to block-parallelism

The four biggest limiting factors for the applicability of block-parallelism are its optimality at small ranges of S, the need for batch processing, the increased space requirement for class II algorithms and the potential complexity of implementations. While changing the optimality at lower ranges of S may be hard, the most obvious directions for future work address the remaining limitations.

In case online applications assume that observations arrive in sufficiently large blocks, all parallelisms may be applied normally. If calculations have to be performed at every t, block-parallelism is not applicable. However, it is important to note that the point of block-parallelism is exactly to perform calculations on blocks without assumptions on the results of preceding and succeeding blocks, which has strong analogy with online algorithms. While online calculations for forward and Viterbi algorithms are trivial, the backward pass and Baum-Welch algorithms may be possible to perform in an online manner using conditionals (see Turin (1998)).

The increased ($\times S$) space requirement may be greatly reduced for class II algorithms by applying checkpointing together with block-parallelism (see below, under Checkpointing).

The complexity of block-parallelism may be a limiting factor in its wide-scale adoption. One important direction for future work would be to implement sufficiently powerful and sufficiently generally applicable parallel libraries (e.g. in CUDA).

Checkpointing for class II algorithms

In the case of class II algorithms, the space requirement of block-parallelism is $\mathcal{O}(S^2L+ST)$, which may be prohibitive for some applications.

Similarly to the checkpointing-like approach used for the Viterbi algorithm in Nielsen and Sand (2011), it may be possible to apply checkpointing together with block-parallelism assuming $L_b > 1$ to reduce space requirements in certain applications (e.g. for the Baum-Welch algorithm).

An important thing to note for checkpointing is that it may also provide performance improvements for class II algorithms for strategies with $L_b > 1$. If merge steps are performed only for block-ends, the performance of the merge steps may be greatly improved (it will be identical to those of class I algorithms). The block-ends may also serve as checkpoints, starting from which the calculations for intermediate time points may be performed in a following (a third, post-processing) step. Naturally, the post-processing step is also an HMM algorithm run, which may be parallelised over the state-space, and calculations over multiple observation blocks may also be performed in parallel in case that is optimal in the available platform.

It is clear that the inclusion of checkpointing will rearrange the performance landscape of block-parallel strategies. For instance, block-parallelism will be able to achieve accelerations on distributed systems for class II algorithms.

For these reasons, checkpointing may be one of the most important possible additions to this work.

Further improvements

Beyond directly addressing the limitations of block-parallelism, there are a collection of further improvements that may also be valuable to achieve.

One option is to develop algorithms and optimisations for multi-level architectures with different communication costs on each level. Another option is to develop block-parallel analogues of modified HMM algorithms (an maybe also include them in a parallel library), such as the lazy Viterbi and simplified or approximate versions of the recursions, etc. Following Turin (1998) it would also be important to provide a full analysis of strategies applicable for the Baum-Welch algorithm.

5.4.2 More rigorous analyses

Undoubtedly, a more rigorous performance analysis may be performed with estimating synchronization frequency for every combination of $W \times S \times T$.

It may also be possible in future performance assessments to account for the differences in calculation requirements between block(s) including t=1 and consecutive blocks in parallel merge and could calculate with different block lengths for the first block (as in Nielsen and Sand (2011)).

For specific applications, one should conduct performance analysies using the BSP model.

5.4.3 Parallelisms for generalised HMMs and Bayes nets

In the case of non-homogeneous and variable-order HMMs, the application of block-parallelism is trivial, with minimal modifications.

In two further classes of generalised HMMs, it is easy to see that block-parallelism may not be the best option for achieving accelerations. In the case of coupled HMMs, the effective state space tends to be large ($S = S_1 S_2$), implying that state-wise parallelisms may allow for sufficient parallelism when the target platforms are appropriate. On platforms with inefficient communications, block-parallelism may still be an option.

In the case of semi-hidden Markov Models (SHMMs), the computation complexity is $\mathcal{O}(S^2 T D)$, where the duration D may be limited or may be as large as T. Unless D is strongly limited, it is likely that SHMM algorithms are best parallelised over the state-space and the space of durations, potentially achieving an $\mathcal{O}(S^2 D)$ parallelism.

It is possible that the conditional block-parallel framework may be applicable to general Bayes nets if the nets are dividable into a sufficient number of sub-graphs with appropriate vertex-cuts that can work as block boundaries. Calculations are then separable into blocks, after which some appropriate merge steps may recover the final results.

6 Conclusions

This chapter introduced a new approach to the theory of block-parallelism and explored the applicability, limitations, relevance and performance of various block-parallelisation strategies.

It has been argued that the conditional approach to block-parallelism may allow for a better starting point for the development of complex applications.

The presented basic block-parallel strategies allow for a wide range of optimisation choices on different platforms. It has been argued that block-parallelism is the only option (beyond trivial parallelisms) to achieve parallel accelerations on systems that do not support efficient communications between parallel threads.

The theoretical performance results indicate that block-parallelism may be applicable for a large range of problem sizes and the optimality of strategies is far from straightforward and potentially requires some analysis from developers ahead of implementations upon adoption.

Following from the applicability and performance results, it is clear that – with sufficient further development – block-parallelism may become a vital component for improving the applicability and power of HMM methods for large problems.

BIBLIOGRAPHY

Emerald: e-Infrastructure South GPU supercomputer. http://people.maths.ox.ac.uk/gilesm/emerald.html.

Molecular biology of the cell. Garland Science, 2008.

Top 500 supercomputers. http://www.test.org/doe/, June 2012.

Kjersti Aas, Line Eikvil, and Ragnar Bang Huseby. Applications of hidden markov chains in image analysis, 1999.

Cornelis A. Albers, Gerton Lunter, Daniel G. MacArthur, Gilean McVean, Willem H. Ouwehand, and Richard Durbin. Dindel: Accurate indel calls from short-read data. Genome Research, 2010.

Michael Bauer, Henry Cook, and Brucek Khailany. Cudadma: optimizing gpu memory bandwidth via warp specialization. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, pages 12:1–12:11, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0771-0.

J. Besag. Spatial interaction and the statistical analysis of lattice systems. Journal of the Royal Statistical Society, 1974.

J. Besag. On the statistical analysis of dirty pictures. JRSS(b), 48:259–302, 1986.

J. Eric Bickel. Some comparisons among quadratic, spherical, and logarithmic scoring rules. Decision Analysis, 4(2):49–65, June 2007.

Rajesh Bordawekar, Uday Bondhugula, and Ravi Rao. Believe it or not! multicore cpus can match gpus for flop-intensive applications! Technical report, IBM, April 2010.

Luke Cartey, Rune Lyngsø, and Oege de Moor. Synthesising graphics card programs from dsls. In Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation, PLDI '12, pages 121–132, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9. doi: 10.1145/2254064.2254080. URL http://doi.acm.org/10.1145/2254064.2254080.

Reed A. Cartwright. Problems and solutions for estimating indel rates and length distributions. Molecular Biology and Evolution, Nov 2008.

Kerrie Nichol Edamura Christopher E. Pearson and John D. Cleary. Repeat instability: mechanisms of dynamic mutations. Nature Reviews Genetics, Oct 2005.

Olivier Delaneau, Jonathan Marchini, and Jean-Francois Zagury. A linear complexity phasing method for thousands of genomes. Nature Methods, 9(2):179–181, 2012.

Olivier Delaneau, B. Howie, A. Cox, Jean-Francois Zagury, and Jonathan Marchini. Haplotype estimation using sequence reads. American Journal of Human Genetics, 93(4): 787–696, 2013a.

Olivier Delaneau, Jean-Francois Zagury, and Jonathan Marchini. Improved whole chromosome phasing for disease and population genetic studies. Nature Methods, 10(1):5–6, 2013b.

Olivier Delaneau, Jean-Francois Zagury, and Jonathan Marchini. Integrating sequence and array data to create an improved 1000 Genomes Project haplotype reference panel. 2014.

Zhihui Du, Zhaoming Yin, and David A. Bader. A tile-based parallel viterbi algorithm for biological sequence alignment on gpu with cuda. In IPDPS Workshops, pages 1–8. IEEE, 2010.

Alex Duval and Richard Hamelin. Mutations at coding repeat sequences in mismatch repair-deficient human cancers: Toward a new concept of target genes for instability. Cancer Research, May 2002.

Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. IEEE Micro, 32(3):122–134, May 2012. ISSN 0272-1732. doi: 10.1109/MM.2012.17.

Claude FÃľrec et al. Gross genomic rearrangements involving deletions in the cftr gene: characterization of six new events from a large cohort of hitherto unidentified cystic fibrosis chromosomes and meta-analysis of the underlying mechanisms. European Journal of Human Genetics, 2006.

Steven Fortune and James Wyllie. Parallelism in random access machines. In Proceedings of the Tenth Annual ACM Symposium on Theory of Computing, STOC '78, pages 114–118, New York, NY, USA, 1978. ACM. doi: 10.1145/800133.804339. URL http://doi.acm.org/10.1145/800133.804339.

Miguel Garcia-Diaz and Thomas A. Kunkel. Mechanism of a genetic glissando*: structural biology of indel mutations. Trends in Biochemical Sciences, 31(4): 206 – 214, 2006. ISSN 0968-0004. doi: DOI:10.1016/j.tibs.2006.02.004. URL http://www.sciencedirect.com/science/article/B6TCV-4JHMHPJ-1/2/36b23428b1616f185977e95860661d75.

Andrew Gelman, John B. Carlin, Hal S. Stern, and Donald B. Rubin. Bayesian Data Analysis. Chapman and Hall/CRC, 2003.

J.Alicia Grice, Richard Hughey, and Don Speck. Reduced space sequence alignment. Computer applications in the biosciences : CABIOS, 13(1):45–53, 1997. doi: 10.1093/bioinformatics/13.1.45. URL http://bioinformatics.oxfordjournals.org/content/13/1/45.abstract.

Garrett Hellenthal, George B J Busby, Gavin Band, James F Wilson, Cristian Capelli, Daniel Falush, and Simon Myers. A genetic atlas of human admixture history. Science, 343(6172):747–51, 2014. ISSN 1095-9203. URL http://www.biomedsearch.com/nih/genetic-atlas-human-admixture-history/24531965.html.

D.R. Horn, M. Houston, and P. Hanrahan. Clawhmmer: A streaming hmmer-search implementatio. In Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference, page 11, nov. 2005. doi: 10.1109/SC.2005.18.

B. Howie, J. Marchini, and M. Stephens. Genotype imputation with thousands of genomes. G3: Genes, Genomics, Genetics, 1(6):457–70, 2011.

B. Howie, C. Fuchsberger, M. Stephens, J. Marchini, and G. R. Abecasis. Fast and accurate genotype imputation in genome-wide association studies through pre-phasing. Nature Genetics, 44(8):955–59, 2012.

Bryan N. Howie and Jonathan Marchini. Genotype imputation for genome-wide association studies. Nature Reviews Genetics, 5(6):499–511, 2010.

Bryan N. Howie, Peter Donnelly, and Jonathan Marchini. A flexible and accurate genotype imputation method for the next generation of Genome-Wide association studies. PLoS Genet, 5(6):e1000529, June 2009. doi: 10.1371/journal.pgen.1000529. URL http://dx.doi.org/10.1371/journal.pgen.1000529.

Richard R. Hudson. Properties of a neutral allele model with intragenic recombination. Theoretical Population Biology, 23(2):183 – 201, 1983. ISSN 0040-5809. doi: 10.1016/0040-5809(83)90013-8. URL http://www.sciencedirect.com/science/article/pii/0040580983900138.

Shawn R. Hymel. Massively parallel hidden markov models for wireless applications. Master's thesis, Virginia Polytechnic Institute and State University, Dec 2011.

National Human Genome Research Institute. A catalog of published genome-wide association studies. http://www.genome.gov/GWAStudies/, 2011.

J.F.C. Kingman. The coalescent. Stochastic Processes and their Applications, 13(3):235 – 248, 1982. ISSN 0304-4149. doi: 10.1016/0304-4149(82)90011-4. URL http://www.sciencedirect.com/science/article/pii/0304414982900114.

David B. Kirk and Wen-mei W. Hwu. Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann, 2010.

Daniel John Lawson, Garrett Hellenthal, Simon Myers, and Daniel Falush. Inference of population structure using dense haplotype data. PLoS Genet, 8(1):e1002453, 01 2012.

Anthony Lee, Christopher Yau, Michael B. Giles, Arnaud Doucet, and Christopher C. Holmes. On the utility of graphics cards to perform massively parallel simulation with advanced monte carlo methods. May 2009. URL http://arxiv.org/abs/0905.2441.

Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In Proceedings of the 37th annual international symposium on Computer architecture, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0053-7. doi: 10.1145/1815961.1816021. URL http://doi.acm.org/10.1145/1815961.1816021.

Jun Li, Shuangping Chen, and Yanhui Li. The fast evaluation of hidden markov models on gpu. In Intelligent Computing and Intelligent Systems, 2009. ICIS 2009. IEEE International Conference on, volume 4, pages 426 –430, nov. 2009. doi: 10.1109/ICICISYS.2009.5357649.

N. Li and M. Stephens. Modeling linkage disequilibrium and identifying recombination hotspots using single-nucleotide polymorphism data. Genetics, 165:2213–2233, Dec 2003.

Yun Li, Cristen J. Willer, Jun Ding, Paul Scheet, and GonÃğalo R. Abecasis. MaCH: Using Sequence and Genotype Data to Estimate Haplotypes and Unobserved Genotypes. Genetic Epidemiol, 34(8), December 2010.

Chuan Liu. cuHMM: a CUDA implementation of hidden Markov model training and classification. 2009.

Gerton Lunter. Probabilistic whole-genome alignments reveal high indel rates in the human and mouse genomes. Bioinformatics, 2007a.

Gerton Lunter. Hmmoc - a compiler for hidden markov models. Bioinformatics, 23(18): 2485–2487, September 2007b.

Gerton Lunter and Martin Goodson. Stampy: A statistical algorithm for sensitive and fast mapping of illumina sequence reads. Genome Research, Oct 2010.

Teri A. Manolio. Genomewide association studies and assessment of the risk of disease. New England Journal of Medicine, 2010.

Jonathan Marchini, Bryan Howie, Simon Myers, Gil McVean, and Peter Donnelly. A new multipoint method for genome-wide association studies by imputation of genotypes. Nat Genet, 39(7):906–13, July 2007.

Ryan E. Mills, Christopher T. Luttig, Christine E. Larkins, Adam Beauchamp, Circe Tsui, W. Stephen Pittard, and Scott E. Devine. An initial map of insertion and deletion (indel) variation in the human genome. Genome Res, 2006.

G. E. Moore. Cramming More Components onto Integrated Circuits. Electronics, 38(8): 114–117, April 1965. ISSN 0018-9219. doi: 10.1109/JPROC.1998.658762.

John Nickolls and William J. Dally. The gpu computing era. IEEE Micro, 30:56–69, 2010. ISSN 0272-1732. doi: http://doi.ieeecomputersociety.org/10.1109/MM.2010.41.

John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. Queue, 6(2):40–53, March 2008. ISSN 1542-7730.

Jesper Nielsen and Andreas Sand. Algorithms for a parallel implementation of hidden markov models with a small state space. In IPDPS Workshops, pages 452–459. IEEE, 2011. ISBN 978-1-61284-425-1.

NVIDIA. NVIDIA CUDA C Programming Guide, February 2014. Version 6.0.

J. O'Connell, D. Gurdasani, Olivier Delaneau, Nicola Pirastu, Sheila Ulivi, Massimiliano Cocca, Michela Traglia, Jie Huang, Jennifer E. Huffman, Igor Rudan, Ruth McQuillan, Ross M. Fraser, Harry Campbell, Ozren Polasek, Gershim Asiki, Kenneth Ekoru, Caroline Hayward, Alan F. Wright, Veronique Vitart, Pau Navarro, Jean-Francois Zagury, James F. Wilson, Daniela Toniolo, Paolo Gasparini, Nicole Soranzo, Manjinder S. Sandhu, and Jonathan Marchini. A General Approach for Haplotype Phasing across the Full Spectrum of Relatedness. PLoS Genetics, 10(4):e1004234, 2014.

J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. Gpu computing. Proceedings of the IEEE, 96(5):879 –899, may 2008. ISSN 0018-9219. doi: 10.1109/JPROC.2008.917757.

Joshua S. Paul and Yun S. Song. Blockwise hmm computation for large-scale population genomic inference. Bioinformatics, 28(15):2008–2015, 2012.

L. Pray. Dna replication and causes of mutation. Nature Education, 2008.

Lawrence Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. In IEEE, volume 77, pages 257–286, 1989.

Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David Blair Kirk, and Wen mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In Siddhartha Chatterjee and Michael L. Scott, editors, PPOPP, pages 73–82. ACM, 2008. ISBN 978-1-59593-795-7.

A. Sand, C.N.S. Pedersen, T. Mailund, and A.T. Brask. Hmmlib: A c++ library for general hidden markov models exploiting modern cpus. In Parallel and Distributed Methods in

Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop on, pages 126 –134, 30 2010-oct. 1 2010.

Matthew Stephens and Paul Scheet. Accounting for decay of linkage disequilibrium in haplotype inference and missing-data imputation. Am J Hum Genet, 76(3):449–62, March 2005.

Matthew Stephens and Paul Scheet. A fast and flexible statistical model for large-scale population genotype data: applications to inferring missing genotypes and haplotypic phase. Am J Hum Genet, 78(4):629–44, April 2006.

Zhan Su. Statistical Methods for the Analysis of Genetic Association Studies. PhD thesis, University of Oxford, 2008.

C Tarnas and R Hughey. Reduced space hidden markov model training. Bioinformatics, 14(5):401–406, 1998. doi: 10.1093/bioinformatics/14.5.401. URL http://bioinformatics.oxfordjournals.org/content/14/5/401.abstract.

The 1000 Genomes Project Consortium. A map of human genome variation from population-scale sequencing. Nature, Oct 2010.

The 1000 Genomes Project Consortium. 1000 genomes project. http://www.1000genomes.org, 2011.

The International HapMap Consortium. A second generation human haplotype map of over 3.1 million snps. Nature, 2007.

The International HapMap Consortium. Hapmap3. http://www.sanger.ac.uk/resources/downloads/human/hapmap3.html, 2011.

Irmtraud M Meyer Tin Y Lam. Efficient algorithms for training the parameters of hidden markov models using stochastic expectation maximization (em) training and viterbi training. Algorithms for Molecular Biology, 5(38), 2010.

William Turin. Unidirectional and parallel Baum-Welch algorithms. IEEE Transactions on Audio, Speech, and Language Processing, 6:516–523, 1998. doi: 10.1109/89.725318.

Ulo Vali, Mikael Brandstrom, Malin Johansson, and Hans Ellegren. Insertion-deletion polymorphisms (indels) as genetic markers in natural populations. BMC Genetics, Jan 2008.

Leslie G. Valiant. A bridging model for parallel computation. Commun. ACM, 33(8):103–111, August 1990. ISSN 0001-0782. doi: 10.1145/79173.79181. URL http://doi.acm.org/10.1145/79173.79181.

Richard Vuduc and Kent Czechowski. What gpu computing means for high-end systems. Micro, IEEE, 31(4):74 –78, july-aug. 2011. ISSN 0272-1732. doi: 10.1109/MM.2011.78.

Richard Vuduc, Aparna Chandramowlishwaran, Jee Choi, Murat Guney, and Aashay Shringarpure. On the limits of GPU acceleration. In HotPar'10: Proceedings of the 2nd USENIX conference on Hot topics in parallelism, page 13, Berkeley, CA, USA, 2010. USENIX Association.

Christopher Yau and Christopher C. Holmes. A decision theoretic approach for segmental classification using Hidden Markov models, July 2010. URL `http://arxiv.org/abs/1007.4532`.

Dan Zhang, Rongcai Zhao, Lin Han, Tao Wang, and Jin Qu. An implementation of viterbi algorithm on gpu. pages 121 –124, dec. 2009.