Parallel Markov Chain Monte Carlo

by

Jonathan Michael Robert Byrd

Thesis

Submitted to the University of Warwick

for the degree of

Doctor of Philosophy

Computer Science

JUNE 2010

Contents

List of Tables

List of Figures

Acknowledgments

This work was funded by and conducted within the Department of Computer Science at the University of Warwick.

I would like to thank all the staff at the Computer Science Department for making this work possible. In particular I would like to express my gratitude to my supervisor Professor Stephen Jarvis for his direction and guidance. Thanks are also due to Professor Abhir Bhalerao for his assistance with the theoretical aspects of the motivational research, and to Dr Elke Thönnes for providing a statisticians point of view of this work . Finally, I would like thank my parents Mike and Gill Byrd for their unwavering support and encouragement.

This thesis was typeset with LaTeX 2$_\varepsilon$[*] by the author.

[*]LaTeX 2$_\varepsilon$ is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society.

Declarations

This thesis is presented in accordance with the University of Warwick regulations for the degree of Doctor of Philosophy. It has been written by myself and has not been submitted in any previous applications for any degrees. The work described in the thesis has been undertaken by myself except where otherwise stated. Portions of this work have been published in the following papers:

- J. M. R. Byrd, S. A. Jarvis and A. H. Bhalerao. Reducing the Run-time of MCMC Programs by Multithreading on SMP Architectures. *22nd IEEE International Parallel & Distributed Processing Symposium (IPDPS'08).* Miami, Florida, USA, 18 April 2008.

- J. M. R. Byrd, A. Bhalerao, S. A. Jarvis. Speculative Moves: Multithreading Markov Chain Monte Carlo Programs. *High-Performance Medical Image Computing and Computer Aided Intervention (HP-MICCAI 08).* New York, USA, 10 September 2008.

- J. M. R. Byrd, S. A. Jarvis and A. H. Bhalerao. On the Parallelisation of MCMC-based Image Processing. *24th IEEE International Parallel & Distributed Processing Symposium (IPDPS'10)*, Atlanta, Georgia, USA, 19 April 2010.

- J. M. R. Byrd, S. A. Jarvis and A. H. Bhalerao. On the Parallelisation of MCMC by Speculative Chain Execution. *24th IEEE International Parallel*

& Distributed Processing Symposium (IPDPS'10), Atlanta, Georgia, USA, 23 April 2010.

Abstract

The increasing availability of multi-core and multi-processor architectures provides new opportunities for improving the performance of many computer simulations. Markov Chain Monte Carlo (MCMC) simulations are widely used for approximate counting problems, Bayesian inference and as a means for estimating very high-dimensional integrals. As such MCMC has found a wide variety of applications in fields including computational biology and physics, financial econometrics, machine learning and image processing.

This thesis presents a number of new method for reducing the runtime of Markov Chain Monte Carlo simulations by using SMP machines and/or clusters. Two of the methods speculatively perform iterations in parallel, reducing the runtime of MCMC programs whilst producing statistically identical results to conventional sequential implementations. The other methods apply only to problem domains that can be presented as an image, and involve using various means of dividing the image into subimages that can be proceed with some degree of independence.

Where possible the thesis includes a theoretical analysis of the reduction in runtime that may be achieved using our technique under perfect conditions, and in all cases the methods are tested and compared on selection of multi-core and multi-processor architectures. A framework is provided to allow easy construction of MCMC application that implement these parallelisation methods.

Keywords: Parallel, MCMC, Markov Chain Monte Carlo, Speculative, Image Processing

Abbreviations

MC	Markov Chain
MCMC	Markov Chain Monte Carlo
MC^3	Metropolis-Coupled Markov Chain Monte Carlo
RJ-MCMC	Reversible Jump Markov Chain Monte Carlo
DR-MCMC	Delayed-Rejection Markov Chain Monte Carlo
MH	Metropolis-Hastings
MPI	Message Passing Interface
SMP	Symmetric MultiProcessing

Nomenclature

γ	In the circle-finding algorithms of section 2.5, this is the exponent applied to the posterior probability to control the 'heat' of the MCMC simulation (the degree to which the transition kernel from section 2.2.4 accepts 'bad' moves). First used on page 31.
λ	The expected number of circles to be found in an image when using one of the circle-finding algorithms from section 2.5. First used on page 28.
ω	In the circle-finding algorithms of section 2.5, this is the exponent applied to the likelihood term to allow the adjustment to the relative importance between the likelihood and prior values. First used on page 31.
ϕ_o	From the circle finding algorithms of section 2.5, this is the contribution to the prior term obtained from evaluating the overlapping circles. It is a term used to penalise circles in close proximity, to avoid the algorithm stacking circles on top of one another. First used on page 29.
ϕ_p	From the circle finding algorithms of section 2.5, this is the contribution to the prior term obtained from evaluating the circle positions. It is a measure of how well the location of the circles in the model match the expected distribution of circles in the image. First used on page 29.
ϕ_r	From the circle finding algorithms of section 2.5, this is the contribution to the prior term obtained from evaluating the circle radii. It is a measure of

	how well the radii of the circles in the model match the expected mean circle radius (r_μ). First used on page 29.
τ	The mean processing time per MCMC move (considering all possible types of move available to the simulation). This includes the time to create the move and determine exactly what it will change, as well as to calculate the prior and likelihood terms and apply the Metropolis-Hastings test to determine if the move is to be accepted or rejected. First used on page 55.
τ_f	The mean processing time per $\mathbf{M_f}$ move. First used on page 56.
τ_g	The mean time required to propose, consider, and apply the Metropolis-Hastings test to a move from the $\mathbf{M_g}$ set. First used on page 78.
τ_l	The mean time required to propose, consider, and apply the Metropolis-Hastings test to a move from the $\mathbf{M_l}$ set. First used on page 78.
τ_s	The mean processing time per $\mathbf{M_s}$ move. First used on page 56.
Θ_{sobel}	The edge orientation map obtained from Sobel filtering an image, eq. (2.11). First used on page 28.
C	A configuration (collection) of circles. Creating a configuration accurately describing the circles in an image is the objective of the MCMC applications in section 2.5. First used on page 26.
c	A circle from a configuration (collection) of circles. These circles are the 'features' being identified in the MCMC applications presented in section 2.5. The centre of circle c is located at coordinates (c_x, c_y) and has radius c_r. First used on page 26.
$G_?$	A Sobel filtered image, the subscript indicating which Sobel filter was applied: G_h is the consequence of applying the horizontal Sobel filter, eq. (2.8), G_v the vertical filter (eq. (2.9)). First used on page 26.

$I(x,y)$	The intensity (average colour value) of the pixel at coordinates (x,y) in a bitmap image. First used on page 26.
K	The number of sample points considered for each circle when determining the likelihood of a configuration when using the first circle-finding algorithm in section 2.5. First used on page 30.
M	The set of all pixels in an image. First used on page 90.
M_{sobel}	The edge magnitude map obtained from Sobel filtering an image, eq. (2.10). First used on page 26.
$\mathbf{M_f}$	A set of MCMC moves that can be processed rapidly, as oppose to moves from set $\mathbf{M_s}$. First used on page 56.
$\mathbf{M_g}$	A set of MCMC moves the acceptance of which impacts the likelihood and/or prior contribution of of the features in the model, as opposed to moves from the $\mathbf{M_l}$ set. First used on page 76.
$\mathbf{M_l}$	A set of MCMC moves the acceptance of which has a spatially localised impact. Unlike $\mathbf{M_g}$ moves the impact on the model's prior and likelihood is limited to impacting only those features in very close proximity to the feature being changed. First used on page 76.
$\mathbf{M_s}$	A set of MCMC moves that take significantly longer to process compared to moves from an opposite set $\mathbf{M_f}$. First used on page 56.
N	The number of iterations to be performed by an MCMC program. First used on page 44.
n	The number of concurrent processes, precise nature of these processes depends on the context in which n is used. First used on page 43.
p_{gr}	The probability that an arbitrary $\mathbf{M_g}$ move will be rejected by the Metropolis-Hastings test. First used on page 79.

p_{lr}	The probability that an arbitrary $\mathbf{M_l}$ move will be rejected by the Metropolis-Hastings test. First used on page 79.
p_r	The probability of rejecting a proposed move in a MCMC simulation. First used on page 43.
q_f	The probability that a arbitrary move proposal will be from the set $\mathbf{M_f}$, thus be a fast-processing move. First used on page 56.
q_g	The probability that an arbitrary MCMC move proposal will be of the $\mathbf{M_g}$ set as oppose to being a member of $\mathbf{M_l}$. First used on page 77.
r_μ	The expected radius of the circles to be found in an image when using one of the circle-finding algorithms from section 2.5. First used on page 28.
r_σ	The expected standard deviation from the mean circle radii (r_μ), when using one of the circle-finding algorithms from section 2.5. First used on page 28.

Chapter 1

Introduction and MCMC Theory

Markov Chain Monte Carlo (MCMC) is a computational intensive technique for sampling from a (typically very large) probability distribution. Algorithms of this class are most commonly applied to calculating estimates for multi-dimensional integrals, and have numerous applications in Bayesian statistics, computational physics and computational biology. Notable and varied examples include constructing phylogenetic trees and other bioinformatics applications [39, 44, 67], spectral modelling of X-ray data from the Chandra X-ray satellite [7], and for calculating financial econometrics [41].

MCMC using Bayesian inference is particularly suited to problems where there is prior knowledge of certain aspects of the solution. For instance, when counting tree crowns in satellite images where the trees will mostly be arranged in a regular pattern [51]. By incorporating expected properties of the solution, the stability of the simulation is improved and the chances of consistent false-positives is reduced. Whilst it can be used to obtain a single model for a dataset in a manner similar to Genetic Algorithms, its true power lies in its potential for evaluating alternative interpretations of the same data. The primary weakness of the method is the time it can take to perform such an analysis. By utilising parallel processing, both on a single machine (multiprocessor, multicore) and by spreading computation across a cluster we can reduce the real-time required to produce results and/or improve the accuracy of the MCMC simulation.

As will be explained in section 2.2.2, Monte Carlo applications are generally considered embarrassingly parallel [53], using two processors will allows samples to be gathered twice as quickly. This also applies for Markov Chain Monte Carlo (discussed in more detail in section 2.2.3, provided the chain(s) have had sufficient time to converge. Unfortunately for high-dimensional problems for which MCMC is best suited, the burn in time required for getting good samples can be considerable. When dealing with very large state-spaces and/or complicated compound states (such as searching for features in an image) it can take a long time for a MCMC simulation to converge on a satisfactory model, both in terms of the number of iterations required and the complexity of the calculations occurring in each iteration. As an example, the mapping of vascular trees in retinal images as detailed in [21, 58] took upwards of 4 hours to converge when run on a 2.8GHz Pentium 4, and takes much longer to explore alternative modes (additional potential interpretations for the input data). The practicality of such solutions (in real-time clinical diagnostics for example) is therefore limited.

If there are multiple credible interpretations for the input data, and these interpretations are not expected to be radically different, duplicating the simulation will not substantially reduce runtime as the time required for convergence will dominate over the time collecting samples. Statistical techniques already exist for improving the rate of convergence, indeed most current optimisation and/or parallelisation strategies take this approach. Whilst some such methods are explained in section 2.4, they are not the focus of this document. The motivation for the work presented in this thesis is to find methods of reducing the runtimes of MCMC applications by focusing on the implementation of on single MCMC chain rather than by modifying the statistical algorithm to improve the rate of convergence. Attempting to achieve parallelisation in this manner is non-trivial as the underlying structure of this class of application - a Markov Chain - is inherently opposed to its calculations proceeding concurrently; by definition the state of a Markov Chain depends only on its preceding state, requiring state changes relating to a single chain to occur in a strictly sequential order[*]. Whilst some of these these new methods increase the amount of work to be performed in absolute terms, the fact that much will be performed concurrently results in a net reduction in runtime. Parallelisation to obtain x times as many samples is trivial, parallelisation within each chain requires more careful examination. Fortunately, since the intent is not to modify the theoretical method by which MCMC works, the methods presented will generally complement and not compete with statistical means of attaining runtime reductions.

1.1 Thesis Contributions

The contributions of this thesis are as follows:

- We propose two new methods (termed 'speculative moves' and 'speculative chains') of implementing Markov Chain Monte Carlo algorithms to take advantage of multi-core and multi-processor machines. Being a purely implementational change the results are unaffected, whilst the runtime of typical MCMC programs can be reduced by $\sim 40\%$ using just two processes.

- We propose a new modification of Markov Chain Monte Carlo termed 'periodic partitioning' that permits conditional parallel processing on a large scale with a limited (and statistically acceptable) impact on the results.

- We propose a number of methods that can be applied to MCMC image process-

[*]The aforementioned methods of $(MC)^3$ and the 'embarrassingly parallel' nature of MCMC both operate by running two (or more) chains in parallel, whereas this thesis seeks to perform parallel processing on a single logical chain

ing problems that reduce the runtime by considering (temporarily or permanently) portions of the image as independent images in their own right. Whilst lacking the statical certainty accompanying the other parallelisation methods presented, the potential runtime improvements are substantially higher whilst giving results that will be reasonable for many applications.

- We fully implement these methods on a number of different machine architectures and demonstrate the suitability of these architectures for these new approaches.

- We provide methods for predicting the runtime of MCMC programs using our speculative moves, speculative chains and periodic partitioning methods, and provide practical examples demonstrating typical runtime improvements that can be expected from the others, therefore providing: (i) increased certainty in real-world MCMC applications, (ii) a means of comparing alternative supporting architectures in terms of value for money and/or performance.

- Finally, we provide a programming framework that automates much of the construction of MCMC programs. When using this framework the parallelisation methods of speculative moves, speculative chains and periodic parallelisation will be automatically made available with no extra work necessary from the the implementer. The usage of this framework is described and demonstrated in the appendices.

The parallelisation methods 'speculative moves' and 'speculative chains' may be used alongside most existing parallelisation and optimisation techniques whilst leaving the MCMC algorithm untouched, and so may safely be used without fear of altering the results. These methods are designed to operate on the increasingly available multiprocessor and multicore architectures. As the technology improves (e.g. by increasing the number of processing cores that are placed onto a single die) the speculative moves and speculative chains methods will yield greater runtime reductions over a wider range of applications. Periodic parallelisation and other image splitting/partitioning techniques are suitable for both multicore/multiprocessor systems and for across a cluster of computers.

1.2 Thesis Outline

This introductory chapter describes the layout of this thesis, its primary contributions, and introduces the terminology that will be used throughout the document. Chapter 2 presents the background research relevant to the contributions of this thesis. This starts with an overview of parallel processing, the ideas and methods underpinning Markov Chain Monte Carlo, followed by the MCMC method itself and a discussion of how and where it may be applied. A summary of the existing methods of improving MCMC using parallel processing is presented with examples, along with an explanation of the conventional means of parallelising the MCMC algorithm and how these methods differ from the novel methods presented in this thesis. The chapter goes on to establish a specific context for the work presented in the rest of this thesis by describing in detail two MCMC applications for the segmentation of circular formations in a bitmap image, and in doing so further explain the details of the most general purpose form of the MCMC algorithm (the Metropolis-Hastings transition kernel). Some simple non-parallel optimisations are also covered here for the benefit of readers implementing their own MCMC application. The example applications shown here also serve as the testbed for the parallelisation methods presented later.

Having provided background and context to MCMC applications in chapter 2, chapter 3 presents the first contribution of this thesis, the parallelisation method 'speculative moves'. Once the rational for this method and the revised MCMC implementation have been explained, a formula for calculating the predicted runtime whilst using speculative moves is constructed. The speculative moves method is then tested on the practical example programs presented in chapter 2 using a number of different hardware platforms, and these results are compared with those predicted from the mathematical formula.

The logical development of speculative moves, termed ‘speculative chains’, is dealt with in chapter 4. Since a mathematical formula describing the benefits of this method would quickly become unmanageably complex when attempting to describe anything but the simplest situations, a simulator is constructed and used to predict the runtimes that can be obtained using speculative chains. As with speculative moves, speculative chains is tested on the practical examples from chapter 2 using a number of hardware platforms.

Periodic parallelisation and a variety of other image-splitting methods are presented in chapter 5. Unlike speculative moves and chains, the methods presented in chapter 5 modify the basic MCMC algorithm in ways that will not be appropriate for all applications. However, with suitable applications, careful implementation and thorough testing these parallelisation methods can produce a substantially larger reduction in runtime that either speculative moves or chains would be capable of.

Chapter 6 concludes the research aspect of this thesis, the developmental work being described in the appendices. The software developed for this thesis consists of a framework with which to construct MCMC applications quickly and efficiently, without the implementer needing to write repetitive boilerplate code. Applications constructed with this framework (termed pMCMC) can implement the three major and new parallelisation methods presented by this thesis with minimal work from the application implementers. An overview of the pMCMC framework and its benefits to any MCMC implementers occupies appendix A. To demonstrate the ease by which fully-featured MCMC applications may be developed using this new framework constructed for this thesis, appendix B contains an example implementation using pMCMC on one of the circle-finding methods from section 2.5. Finally the usability of the applications built with pMCMC is shown in appendix C, where an example of how end-users interact with a pMCMC program at runtime is provided.

1.3 Terminology

The remainder of this introductory chapter contains an explanation of the terminology that will be used throughout this report. The terminology considered is categorised as concerning *parallel processing*, the *hardware* that is available and that relating to the *image processing* aspects of this work.

1.3.1 Parallel Processing

Thread A *thread of execution* is a sequence of instructions in a computer program that will be carried out sequentially*, but that may be performed concurrently with other threads.

Multithreading The use of multiple threads in a computer program to perform multiple operations simultaneously. Depending on the hardware and decisions made by the operating system the multiple threads may genuinely be performing concurrent operations, or the appearance of simultaneous processing may be created by interleaving the execution of the threads.

Mutex Safe communication between threads requires ensuring certain instructions are carried out in a precise order, for instance one thread should not attempt to read from a shared buffer until another thread has written valid data to said buffer. A mutex is the primitive construct used to ensure this by declaring certain blocks of code to be **mut**ually **ex**clusive. The first instruction in such a code block is to obtain a 'lock' on a mutex, once the mutually exclusive code has been completed the mutex lock is released. At most one thread may possess a specific mutex-lock, any other attempts to obtain a locked mutex result in a failure code, or require the thread to wait until the mutex-lock becomes

*with the exception of any methods applied at the hardware level i.e. pipelining available. The unnecessary use of mutexes is to be avoided as the acquisition and releasing of a mutex lock is a comparatively expensive operation, not to mention the time spent waiting for a mutex lock to become available.

Interleaving The process by which multiple threads may be executed concurrently (at least from the point of view of the end-user) on a single-processor system. Only one thread is actually being executed at any one time, the processor rotates through all the threads that are present. Since each thread receives processor time for only a fraction of a second at a time this creates the illusion of simultaneous execution over human timescales. Unlike true parallel processing, thread interleaving does not reduce the total time required to perform the work placed on the threads, in fact the total required will be longer than if each thread was run to competition sequentially (due to the overhead imposed by switching active threads). This interleaving of the instructions from a number of threads must be used whenever there are insufficient processors to run all the requested threads.

Overhead In this document, overhead will refer to the time added to the program's runtime in order to implement some parallel processing methodology. This includes the time to create/destroy any additional threads that may be needed and (more importantly) the time used in communicating between threads, primarily spent on synchronisation using mutexes but in the case where threads are located on physical distinct components (i.e. multiprocessor architectures or on different nodes in a cluster) the time to physically send a message between them may be significant.

1.3.2 Hardware

Before embarking on the detailed description of the MCMC method and the parallelisation mechanism that have been developed, let us examine the available resources. Computers have long been capable of multitasking, appearing to perform multiple actions at the same time by interleaving the instructions of two or more processes. Over the years a number of methods have been devised to truly perform more than one action at a time ranging from the obvious (use more than one computer) to the subtle (internally duplicate key processing components). In this document we will consider only the following, widely available and economical parallel processing architectures:

Multiprocessor A computer architecture where multiple CPUs access the same shared main memory. Each processor is located on a separate die (or 'chip').

Multicore A more recent innovation, a multicore architecture is one where multiple processing units ('cores') are located on the same die. The cores may share a level of cache, but otherwise are equivalent to CPUs.

SMP Symmetric MultiProcessing is a computer architecture where two or more identical processors or cores are connected to a single shared memory. In this document it is used as a blanket term encompassing both multiprocessor and multicore hardware.

Cluster Multiple computers (termed 'nodes') connected together by a local area network. For the purposes of predicting and testing performance of parallelisation methods, a homogeneous cluster is assumed.

Multicore machines will offer the fastest level of inter-process exchange of information as the processing cores are on the same physical die, along with a shared memory cache. Multiprocessor machine processors will take longer to exchange information as there is greater physical separation between the processors and they must exchange information through slower forms of shared memory (on-board rather than on-die cache, or possibly directly through main memory). Distance between nodes is greatest in a cluster thus communicating between such nodes is the slowest, compounded by the lack of naturally shared memory. The speed of communication within a cluster depends on the quality of the communication channel used to connect the nodes.

1.3.3 Image Processing

While Image Processing is a large field, in this document we concern ourselves only with those problems that can be solved by MCMC - primarily segmentation problems.

Partitioning In this document 'partitioning' will be used to refer to the practise of splitting a large image into smaller components upon which additional processing will take place with some degree of independence.

Modes When analysing complex images it is frequently the case there are multiple possible interpretations that maybe reached from a single set of input data. A cluster of pixels may be interpreted as a single shape, or multiple overlapping shapes, or a meaningless blob that should be ignored. These different interpretations are refered to as multiple *modes*. The Markov Chain Monte Carlo method is particularly good at identifying such modes, and in favourable circumstances assigning relative probabilities to the accuracy of each mode.

Segmentation The partitioning of an image into multiple segments based on their visual characteristics. For example identifying those pixels that comprise any circular structure in an image, as is done in the case studies/motivational research used in this document - see 2.5).

Statespace The statespace of a simulation is the set of all states it is possible for that simulation to be in. It is frequently envisaged as a landscape, each unique state representing a location on the 'ground', with the ground's altitude representing the likelihood or desirability of that state. Iterative methods explore the statespace in search of the highest peaks in the landscape (representing the target/most desirable states) whilst avoiding valleys (‘bad’ states that are less favourable that those around them).

Mixing The ability of a statespace exploring algorithm to explore the entirety of the statespace. A program that displays poor mixing will be unlikely to traverse valleys in the statespace, thus tend to become stuck at local optima - the low peaks of the statespace that serve as a distraction from the actual target state(s), the highest peak(s). Even if it can be shown the whole statespace will be explored given infinite time, a simulation with poor mixing will spend excessive time in local optima states rather than exploring and locating the globally optimal solution(s). Improving the mixing of an MCMC algorithm has the practical effect of reducing the realtime required for it to converge on a acceptable solution my allowing the chain to more rapidly escape low peaks and traverse valleys in the statespace.

Chapter 2

Background and Motivational Research

The purpose of this chapter is to impart an understanding of how the MCMC method works, the significance of MCMC, and the challenges associated with trying to reduce the runtime of MCMC applications through the use of parallel processing. This chapter starts with a general introduction to the parallel processing, then explains the mathematical methods upon when MCMC is based before describing the MCMC method itself. The existing variants of MCMC that involve parallel processing are then examined. Section 2.5 establishes a specific context for discussions throughout the rest of the thesis by presenting two MCMC algorithms capable of identifying and describing circular features in bitmap images. The two algorithms are distinguished by the means of which a circle is identified, one uses an edge detection filter and seeks to locate the thin band of pixels with the magnitude and orientation that denote a circular edge, whilst the other seeks out grouped pixels of high intensity. Both methods come with advantages and disadvantages that makes them suitable for processing different types of image. They also have different memory access profiles, though this did not appear to be significant effect.

2.1 Introduction to Parallel Processing

Parallel processing is a means of reducing the real-time needed to perform some job by splitting that job into smaller tasks and performing at least some of those tasks simultaneously. The total amount of work done will not have decreased, but instead will have been shared over multiple processors. Parallel processing has two prerequisites, the algorithm must contain tasks that can be safely performed concurrently, and the appropriate computer hardware must be available to allow those tasks to be done concurrently without the delays imposed by performing the parallel processing overshadowing the real-time reduction achieved by the tasks running concurrently.

Some applications are very easy to parallelise (split into tasks that can safely be performed in parallel). Consider searching through a large, rarely modified, database for some unsorted, unindexed data. It is a simple matter to speed up the search by halving the database and assigning a different processor to search each half. Indeed, the database can be split into as many sections as there are available processors with no complications. Each database section can be considered independantly, and there is no need for the processors working on separate database sections to communicate. Applications that can be trivially parallelised to an arbitrary extent in this manner are termed embarrassingly parallel.

Parallelisation is harder when there are dependencies between the tasks, as then some tasks must then respect a strict ordering and communicate information between themselves. The purpose of this thesis is to find ways of parallelising a Markov Chain (see section 2.2.1 below), at first glance an impossible task as each state in chain is dependant on its preceding state, leaving no room for parallel processing. Latter chapters will demonstrate that this initial assessment is not entirely correct, that there are some tasks that can be safely performed concurrently.

2.1.1 Parallel Processing Architectures

Assuming tasks that may be performed concurrently in an algorithm have been identified, how is the concurrent execution of those tasks achieved? The type of parallel processing architecture that is appropriate is determined by the frequency with which the parallel processors will need to communicate and share their data. Any form of inter-process communication will take time to perform, thus the implementation of a parallelisation scheme will spend some time making the parallel processing system work rather than working on the problem directly, this is termed overhead. A parallelisation method will only provide net benefits if the reduction in program runtime achieved by the concurrent execution is greater than the overhead incurred in performing that parallel processing.

Since embarrassingly parallel problems can be divided into a great many tasks that require little to no communication between them to execute, it is feasible to distributed these tasks to remote computers and collect in the end results with minimal concern for either the communication system used or specifications of the computer(s) doing the work. An early pioneer of this was the SETI@home project [66, 5], which used idle time on the computers of volunteers from around the world to conduct a search through radio telescope data (downloaded via the internet) for radio signals from extraterrestrial civilizations. More recently efforts have been made to develop a more structured means of farming out loosely coupled parallel processing tasks to available computers around the world, this is called GRID computing. A number of middleware systems have been developed to facilitate GRID computer, including the Globus Toolkit [25] and Berkeley Open Infrastructure for Network Computing (BOINC) [4].

Parallel algorithms that are not embarrassingly parallel will require much tighter coupling (i.e. more communication) between the tasks being performed, To obtain a processing environment where communication between tasks/processes is more reliable and prompt, a group of computers can be linked together by a network connection and used as a single computing resource called a *cluster*. Each constituent computer in a cluster is termed a *node*. Software tools such as Condor [56] and MPI (see section 2.1.2 below) exist to support and automate many of the details of maintaining and using a cluster. Note that entire clusters can also be made available via GRID middleware, GRID is not restricted to to the farming out of embarrassingly parallel applications.

For some applications even cluster computing imposes unacceptable communication overheads. To eliminate the cost incurred in communicating between cluster nodes, a computer can be constructed with multiple CPUs (central processing units) using the same main memory and controlled by a single operating system. Although the CPUs share main memory, each processor will maintain its own internal cache. One step on from multiprocessor computers is to build a single processor that contains multiple processing cores. With both cores on the same die (integrated circuit/chip) they may share the same on-board memory cache and communicate even faster than when the processors are on different chips. Both multicore and multiprocessor computers are referred to a Symmetric MultiProcessing (SMP) systems.

Despite the reductions in inter-processor communication overhead in SMP systems, clusters still have an advantage in that they can easily be extended by the addition of more nodes (computers). An SMP system will generally have a fixed number of processors/cores, and whilst off-the-shelf computers with 2 or 4 processors/cores are becoming the standard commercially available PC, SMP systems with a larger number of processors remain expensive. Clusters in contrast can scale almost any size at a fraction of the cost (especially if discarded, budget, or bulk-brought computers are used as the cluster's nodes).

2.1.2 Inter-process communication

Communication between processes can take two forms. In one, all communication is explicit: process A sends a message to process B, process B at some point waits until a message from process A is received, etc. This is called 'message passing', and is the natural form of communication between physically distinct computers communicating over a network. The information communicated could be brief (just the existence of the message) or carry some payload of data, and both the receiving and sending of messages can be either synchronous (the process will wait until the message is sent/received) or asynchronous* (the process will not wait, if a message cannot be send/received immediately the process continues with other work). As one would expect there are programming aids to automate much of this communication, the standard being Message Passing Interface (MPI) [24], of which LAM/MPI [8, 55] is but one implementation

The alternative to message passing is 'shared memory'. All the processes share a common memory that they can read and write to. Communication between processors is implicit, taking the form of reads/writes of memory locations rather than the explicit packaging and sending of information. In a shared memory scheme care must be taken to ensure that processes do not destructively interfere with each other's work by making unexpected changes to memory in use another process, a concept further explored in section 2.1.4. Multicore and multiprocessor systems naturally implement this scheme as they both share access to the computer's main memory (though in practice each processor will still retain its own local cache to speed up memory operations). Shared memory can also be simulated between distinct computers (i.e. across the nodes of a cluster) by the use of software such as OpenMP [14, 26].

The practical tests used throughout this thesis were conducted primarily on

*When asynchronous message passing is used (either by the sender or the receiver), a buffer is needed to store the transmitted information until the other process is ready to accept it.

multicore and multiprocessor architectures, thus used the shared memory model. It is nonetheless sometimes helpful to refer to the parallel algorithms using message passing terminology to emphasise and clarify how and why the parallel processes are interacting.

2.1.3 Threads

When implementing a parallel program in a multiprocessor or multicore system, each linear sequence of instructions is referred to as a *thread* [13, 48]. A multi-threaded program is one where there are several threads of execution proceeding in parallel. The mapping of threads to physical processors is performed by the underlying operating system, which will try to balance the threads on the processors such that all the processors are equally utilised. Threads run on separate processors (or processor cores) will be truly executed simultaneously, however if there are more threads than processors some processors may be assigned more than one thread. In this case the computer must still give appearance of simultaneous execution even if the actual time taken to perform tasks on both threads is not reduced. Multiple threads on a single processor are therefore executed using time-slicing. Processor time is divided into brief slices, and for each slice control of the processor is rotated to a new thread. The instructions from each of the threads are interleaved in some manner, provided this happens sufficiently frequently these threads will appear to be executing simultaneously (albeit slower than if each thread was the only one on its processor).

The parallelisation methods proposed throughout this thesis depend upon the threads being mapped to separate processors, the parallelisation methods only work if two threads do achieve twice as much work as one thread, in a set period of time. For this reason, unless explicitly mentioned it will be assumed that the operating system has assigned every thread to its own unique processor/processing core, the term 'thread' may well be used interchangeably with 'process'. Experimental results will not be obtain for situations where there are more threads than processors.

2.1.4 Mutual Exclusion

Regardless of whether threads are interleaved or actual parallel processing is taking place, interactions between threads need to be carefully controlled. Although each individual thread will be performing a linear sequence of instructions, the exact order in which instructions from several threads will be performed in is non-deterministic. This is irrelevant if each threads actions will not impact any of the others, but critically important if the threads access some shared resource, i.e. multiple threads attempt to read to/write from a single location in memory as a means of communication. Unless the order in which instructions on that memory location are performed are strictly controlled, the end result will be unpredictable (one thread may read data before, after, or even in the middle of another thread changing that data).

The interactions between threads is controlled by establishing synchronisation points (where threads wait for each other to reach a certain stage in their instructions before either proceeds) and zones of mutual exclusion (blocks of code that at most one thread may be executing at any one time). These are implemented by an operating system primitive called a *mutex* [13, 49]. A mutex may be in one of two states, locked or unlocked. A thread may attempt to obtain a lock on a unlocked mutex, in which case the mutex is moved into the locked state until that same thread explicitly releases the mutex lock. If a thread X attempts to obtain a lock on a mutex that is already locked by a different thread Y, thread X will be blocked (rendered inactive) until thread Y releases its lock on the mutex. Thread X is then reactivated any will obtain mutex for itself before continuing.

The use of a mutex can cause code blocks on different threads to be mutually-exclusive with one another, thus protecting a shared resources (such as a memory location) against the consequences of unintended interleaving of instructions from different threads. The price for this is that mutex operations are more expensive (in terms of time) than many standard actions the thread can perform as the system much check that no other threads are also to trying to access the mutex (mutex operations are 'atomic', there is no possibility of a mutex being half-locked, of multiple threads obtaining the same mutex lock at exactly the same time). In a 'thread-safe' program each thread will work using its own 'local' area of memory that only it will read/write to. Any access to memory locations that multiple threads may use needs to be protected via mutexes. It is safe to have multiple threads simultaneously reading from the same block of memory, but any changes to that memory location should be mutually exclusive with any other reads or writes.

A multithreading structure known as a *condition variable* [49, 48] enables a thread to be kept blocked (inactive) until some condition is met. The classic example of this is the 'bounded buffer problem'. A producer thread wishes to communicate information to a consumer thread, using some fixed capacity buffer (memory location). Producer thread should not write further information to the buffer if it is already full, whilst the consumer thread should not attempt to retrieve information from a buffer that is empty. The threads need to block (wait) on a condition (whether the buffer is empty or full) in addition to the buffer read/write operations being mutually exclusive.

One of the potential dangers from using mutexes is that it is possible to have a system where all threads are waiting for another thread to release a mutex lock, thus no thread is able to proceed. Such a state is called a deadlock. Programmers of multithreaded programs need to be very careful to structure threads interactions to avoid the possibility of deadlocks. For this reason commands to terminate a thread prematurely (rather than waiting for the thread to terminate itself i.e. by running out of instructions to perform) are rarely used, if they are even supported by the programming language/operating system. Forcibly terminating a thread will leave any mutexes it had locked permanently stuck in a locked state. The 'safe' alternative is to have the thread periodically check a 'flag' variable in memory. If the thread is to terminate itself this flag will be set, in which case the thread can release any mutex locks it has an self-terminate in an orderly fashion.

2.1.5 Pipelining

On an even smaller scale, the processors/processor cores in modern systems perform instructions using a technique called pipelining [50, 23]. In a pipeline processor the basic instructions the CPU can process are internally implemented as series of small self-contained modules all of which can perform their task in the same, fixed, period of time. A single CPU operation is then not processed all at once, but rather as the cumulative effect of a sequence of modules: a pipeline. An operation to perform is supplied to the module at the input end of the pipeline. At every clock tick each module passes its result to the next module in the pipeline. Eventually the end result of the original instruction is obtained at the far end of the pipeline. Whilst a single operation may in fact take longer to pass through a pipeline that it would to calculate directly on a non-pipelined processor, a pipeline processor has the capability to achieve a much higher throughput (more operations performed per unit time) as a new instruction can potentially be input (and the result from an earlier instruction extracted) at every clock tick, corresponding to the time taken by the slowest module in the pipeline.

2.2 The Markov Chain Monte Carlo Method

To understand the MCMC method one must first have knowledge of the two mathematical concepts upon which it is based, the idea of a Markov chain and the Monte Carlo method of problem solving.

2.2.1 Markov Chains

A Markov Chain is a sequence of states describing a random walk through some statespace, with the property that the next state in the sequence is dependant only on the current state of the sequence, the preceding states being irrelevant. To put it more formally, it is a discrete random process with the property that the probability of the next state given its past history is the always the same as the probability of the next state given only the current state (the next state is conditionally independent of the past states). If X_1, X_2, $X_3 \ldots$, is a random sequence with the condition that

$$P(X_{n+1} = x | X_1 = x_1, X_2 = x_2, X_3 = x_3, \ldots X_n = x_n) = P(X_{n+1} = x | X_n = x_n) \tag{2.1}$$

then this sequence is a Markov Chain [35]. The set of all possible values of X is called the *state space* of the chain.

Since the determination of the next state of a Markov Chain is random, it is not possible to predict the exact state of the chain in the future, however under certain conditions it is possible to determine some long-term statistical properties about the chain. Specifically, if probabilities governing the transitions between the various states do not change with time (are time homogeneous) then a single matrix $\mathcal{P}$ can be used to represent the probability of every possible state transition. If a vector π exists such that its entries sum to 1 and

$$\pi = \pi \mathcal{P} \tag{2.2}$$

then π is termed the chain's 'stationary distribution'. Intuitively eq. (2.2) states that if a Markov Chain is in the stationary distribution at step t then it will remain in the stationary distribution at step $t + 1$ (the application of transition as governed by $\mathcal{P}$ does not change the distribution the chain is in). π is thus a probability distribution representing the 'steady state' behaviour of the chain. Provided that it is possible to reach any state in the statespace from any other state (the Markov Chain is irreducible) and that the chain is aperiodic (does not contain any states that can only reoccur at regular intervals), then the relative frequency of the states occurring in the Markov Chain will tend towards this distribution irrespective of its starting state, though it may take a large number of state transitions before sampling from the chain does becomes representative of the stationary distribution [47]. When there are sufficient states for this to occur the chain is said to have reach equilibrium, or *converged* on its stationary/equilibrium distribution. Random sampling from the states of a converged Markov Chain is equivalent to sampling from its stationary distribution directly.

2.2.2 Monte Carlo Methods

The Monte Carlo method is a broad class of computational algorithm that generates random samples and observes that some fraction of those samples obey some property or properties [52]. Because of the random element, Monte Carlo algorithms are typically employed to solve problems that are too complicated to solve analytically. A good example (and one of the most common applications) of Monte Carlo is that of Monte Carlo Integration. To integrate a function over some complicate domain D random samples are taken from a simple domain D' such that D' is a superset of D. Samples are tested for membership of D, and the area of D estimated to be the area of D' multiplied by the proportion of sample points that were within D.

Other Monte Carlo application operate in a similar way, using random samples from some simple distribution then applying some testing/processing to get around the difficulty in sampling or directly analysing some complicated distribution. When using any such Monte Carlo method it is important to obtain a large number of samples, as the more samples that are used the more accurate/detailed the end result will be. Since samples are taken entirely independently of each other, Monte Carlo simulation can be termed embarrassingly parallel [53, 17, 59]. It is entirely possible to have multiple processors (threads, processors, or entire machines) working independantly to produce samples. In most cases it will be possible to conduct the bulk of the calculations combining the results of the sampling on these separate processes, leaving the final merging of each process's work a comparative trivial operation. In this setup, the number of processes that can be performing and analysing samples is almost limitless, with each additional processor increasing accuracy or reducing the runtime (depending which is deemed more important).

2.2.3 Markov Chain Monte Carlo

When attempting to sample from a small target distribution within an extremely large and complicated state space the basic Monte Carlo method is simply not practical. The state space is so large that (within any reasonable timescale) too few samples will be found to be within the target distribution to perform any meaningful analysis. This frequently occurs when a state have many degrees of freedom ('variables' that may be altered), as each degree of freedom is equivalent to an additional dimension in the state space and thus responsible for an exponential increase in the volume of the state space. For example, image analysis problems will often suffer from this as each new 'feature' to identify in the image will have its own set of variables characterising it (its x and y coordinates, size, shape etc.).

Fortunately in many such cases, having obtained one state from within the target distribution it is often easy to obtain a second state by making a small change to the first. Using this property we can construct a Markov Chain that performs a random walk through the statespace, each state being a small modification of its predecessor. By constructing the chain's transition kernel such that its stationary distribution is equal to the desired target distribution then sampling from the Markov Chain would be equivalent (long term) to sampling from that target distribution, despite the correlation between consecutive states in the chain. For a detailed examination of this Markov Chain Monte Carlo (MCMC) method the reader is referred to the work of P. Green of the University of Bristol [32, 31] or books such as [52, 29]. Here we provide a summary of what the algorithm does in practice and why.

From the implementers perspective a program for sampling from such a chain is an iterative simulation. At each iteration of the simulation a transition is proposed to move the Markov Chain from state x to some state y by making some small alterations to x. The probability of applying this proposed move is calculated by a transition kernel constructed in such a way that the stationary distribution of the Markov Chain is that of the target distribution. The construction of a suitable kernel is often surprisingly easy, and is frequently done by applying Bayesian inference [33], as described in the following section (section 2.2.4).

Over a (large) number of iterations the chain will conduct a random walk across the statespace, eventually converging on the chain's equilibrium/stationary distribution (that we have arranged to be the same as the posterior distribution). Following convergence, sampling from the Markov Chain will produce models with a frequency proportional to the model's relative posterior probability (the most probable models for the input data and prior knowledge provided will be the most frequently found amongst the samples). Although the chain will converge on the stationary distribution irrespective of its initial state, this may take many iterations. Once a Markov Chain is underway it must be left for a suitable 'burn in' period before samples are taken from it, to allow the chain to converge. Premature sampling will result in sub-optimal models being obtained. Determining how many iterations will be required for convergence (how long the 'burn-in period needs to be) is in general an unsolved problem and beyond the scope of this thesis.

In some applications, typically those dealing with very complex high-dimensional states such in image processing problems, a single near-optimal sample of the target distribution may be all that is required. Even then, taking and comparing many samples allows different 'modes' to be identified in the results - potential alternative interpretations of the same data (e.g. does a particular mass of colour in an image represent a single large blood cell or two blood cells that are overlapping). Unlike most of the alternatives methods of image processing (Genetic Algorithms, and less general-purpose deterministic methods), MCMC can not only identify various possible interpretations for ambiguous data, but also give comparative probabilities for those interpretations, determined by the frequency with which the modes (interpretations) are sampled.

MCMC algorithms with complex and/or high dimensionality states and spanning large state spaces face three practical obstacles that must be overcome.

1. The long time it takes for a complex or large MCMC simulation to converge to its equilibrium position, and then to explore enough of the statespace for sampling to detect and assign probabilities to the alternative modes.

2. Determining when the simulation has reached equilibrium, thus when sampling should start.

3. Determining how many samples will be required to fairly explore nearby modes, once equilibrium has been reached (should more than one sample be required).

Addressing the first obstacle is the purpose of this document. The remaining obstacles are, in the general case, unsolved and beyond the scope of this thesis, though in practise obstacle (3) can be answered 'as long as you can afford' (the more samples gathered the more accurate the final result). As for (2), there have been a number of attempts at obtaining theoretical convergence bounds, though these are often two broad to be of use. There are also a number of methods for detecting convergence by apply diagnostic tools to the outputs of the samplers, though even when using a combination of these is its not possible to say with certainty that a finite sample from an MCMC algorithm is representative of an underlying stationary distribution[15]. For practical applications with consistently similar or predictable datasets the convergence point can simply be estimated from comparisons with established solutions and the assumption of similarity amongst input datasets (i.e. it took around z iterations for the last 10 datasets to converge, so the same will be assumed for the 11th).

2.2.4 Bayesian Inference and the Metropolis-Hastings Method

The standard transition kernel (the algorithm for deciding the probability by which a proposed state change is accepted and applied) used in MCMC is termed Metropolis-Hastings and was proposed in 1970 in [37] as a development of an earlier technique from [46]. To present this in a context that is consistent with that of later chapters the algorithm will be summarised in the form used to perform Bayesian Inference for the purposes of image processing; the construction of a model M to describe target features in some bitmap image I. In other words, the MCMC algorithm will be presented as it applies to selectively converting an image from a bitmap representation I to a vector representation M containing only those image features of interest. It should be noted that the construction of MCMC algorithms for performing specific tasks is a research topic in its own right, and beyond the focus of this thesis. Readers are advised to consult the referenced articles for a complete understanding of the MCMC method and its variants. For the purposes of understanding the parallelisation methods proposed in this thesis, only a knowledge of the mechanical implementation and statistical constraints placed upon that implementation is required.

Bayesian inferences is a means for deriving conclusions using observations to establish or update the probability that a hypothesis is true. Bayes Theorem expresses the relationship between a a conditional probability and its inverse. In the context of finding a model M for image I, it shows how the probability model M is correct given the image I can be expressed in terms of the probability of having image I assuming it is a representation of M.

$$P(M|I) = \frac{P(I|M)P(M)}{P(I)} \tag{2.3}$$

$P(M|I)$ is the *posterior* probability, the probability a model is 'correct' for a given image. P(M) is called the *prior* (or *marginal* probability, the probability of a model being correct without knowing the specific image data. The prior term evaluates how well the model compares to what logical data is 'known' or presumed about the image, the expected number and distribution of features for instance. $P(I|M)$ is the *likelihood* of model M, the term that evaluates how well the data fits the model (the probability of image given a specific model). This is not to be confused with the *posterior* probability (the probability the model is correct given the actual input data). The likelihood is implemented by considering only those features already in the model and determines how likely it is that the features are present in the image data as described by the model. $P(I)$ is the prior probability of the image data, in practical terms this just acts as a normalising constant[*].

Assuming suitable definitions for the prior and likelihood terms are chosen, the 'best' model is one that maximises the posterior probability (the probability of model M given image I). Determining the absolute probabilities is not possible without already knowing the target model (which, if known, renders the matter irrelevant) but it is often feasible to calculate a probability density, a measure of the relative probability of a state but lacking the required normalisation constant to turn this into a true probability. The Metropolis-Hastings method gets around this by constructing a Markov Chain with a model as its state and its equilibrium (stationary) distribution equal to the posterior distribution. The transition kernel deciding the probability with which a transition to a new state M' from the current state M is accepted is based on the ratio between the posterior probability of the new state and old state so that the (unknown) normalising constant $P(I)$ cancels out:

$$\frac{P(M'|I)}{P(M|I)} = \frac{P(I|M')P(M')}{P(I)}\frac{P(I)}{P(I|M)P(M)} = \frac{P(I|M')P(M')}{P(I|M)P(M)} \tag{2.4}$$

The normalising constants in the prior and likelihood probability functions can also be cancelled out, permitting these to be calculated on an arbitrary scale rather than

[*]The use of the term 'image' here is merely for consistency with all the examples used throughout this thesis, I may be any form of data for which we are attempted to describe by some model M.

requiring them as true probabilities:

$$\frac{\text{prior}(M')\text{likelihood}(M'|I)}{\text{prior}(M)\text{likelihood}(M)} \tag{2.5}$$

The Metropolis-Hastings test actually utilised also requires the probability of proposing the move transition $M \rightarrow M'$ and its inverse $M' \rightarrow M$. Additionally if the move involves a change in dimensionality (the number of possible variables that may be changed, for instance if a move adds or removes a feature of the model rather than simply altering it) the Reversible-Jump Metropolis-Hastings variant (proposed by P. Green at the University of Bristol) must be employed, imposing some additional constraints on the potential moves and inserting a Jacobian term into the acceptance test to compensate for the change in dimension [32]. Taking all this into account, given a proposed move generated from a move type chosen at random from a selection of move types (in our image example an instance of one of add, delete, move or change a feature in the model) that would take the simulation's state from M to M', the probability that the move will be applied and M' be the next state is given by:

$$min[1, \frac{\text{prior}(M')}{\text{prior}(M)} \times \frac{\text{likelihood}(M'|I)}{\text{likelihood}(M|I)} \times \frac{p(M', M)}{p(M, M')} \times \mathcal{J}] \tag{2.6}$$

where $p(M, M')$ is the probability of proposing the move from model M to model M', and the Jacobian term ($\mathcal{J}$) is defined by how the dimensionality of the model would change, see [32]. This kernel will produce the probability for advancing the chain to state M' from M based on how well M' fits with the *prior* knowledge (what properties the target configuration is expected to have) and the *likelihood* of M considering the actual data available. Moves that appear to be favourable compared to the current state of the chain have acceptance probabilities > 1 and so are accepted unconditionally, whilst moves to apparently worse states will be accepted with some reduced probability. Once the move/transition has been either accepted (causing a state change) or rejected (leaving the chain's state unchanged) the next iteration begins and a new move is proposed.

Over a (typically large) number of iterations the chain will eventually converge to its equilibrium distribution. As explained earlier, at this point, taking samples from the chain is equivalent to sampling from the target distribution. The probability of a certain state being sampled will be proportional to the posterior probability of that state, thus samples can be expected to be clustered around the models with the maximum (or at least, a locally maximum) chance of matching the input image. When used in practice the time the chain is left to converge *before* taking and using samples from it is called the burn-in time. Although there are a number of method of determining how long this ‘burn-in period’ needs to be [15], in the general case determining the required burn-in duration is an unsolved problem beyond the scope of this thesis.

2.2.5 Delayed Rejection MCMC

One variation of MCMC is known as delayed-rejection MCMC (DR-MCMC). First proposed by L. Tierney and A. Mira (University of Minnesota) [60] and then generalised by P. Green (of Bristol University) in [34]. DR-MCMC seeks to reduce the probability of iterations that do not advance the state of the chain. If a move proposal is at first rejected, a second-stage proposal is attempted that may optionally depend upon the rejected move. This improves performance of the sampler, but at the cost of increased computation per iteration (at least, those iterations that initially reject a transition). Performance is improved by enhancing the efficiency of the statistical algorithm by using rejected moves to improve the probability of the next proposal being accepted.

Delayed-rejection MCMC does not have a natural synergy with the parallelisation methods covered in chapters 3 and 4 as both techniques seek to reduce the realtime ‘wasted’ by MCMC iterations that reject the proposed state change. DR-MCMC sees rejected moves as an opportunity to inform and improve the next move proposal that is made, whereas with speculative moves (the subject of chapter 3) rejected moves are discarded in favour of any accepted speculative moves that were considered in parallel with the rejected move. Whilst it is possible to apply the chapters 3 and 4 parallelisation methods to delayed rejection moves (a single speculative move would consist of both the first move and its second-stage calculations and move proposal) the parallelisation would bring much reduced benefits compared to parallelising normal MCMC as the presence of second stage moves would lower the move-rejection rate (see section 3.3 for the consequences) and lengthening the time per rejected move (see chapter 4 for the problems this may cause). DR-MCMC *can* be used in conjunction with the parallelisation methods from chapter 5 without any problem, as the parallelisation methods in that chapter are based on partitioning the image/data, not paralellising on the level of individual moves.

2.3 Applications of MCMC

MCMC plays a key role in many important fields, particularly bioinformatics and statistical physics . It lies at the heart of the tradition of 'simulation physics'

To provide an understanding of the areas that may benefit from the parallelisation methods proposed in this document this section will summarise a few of the interesting and challenging MCMC applications that have been developed in recent years.

Unlocking the full amount of information contained within the raw data obtained by the Chandra X-ray telescope requires subtle statistical analysis. One goal is to model the distribution of high energy photons from a particular astronomical source. Although simpler algorithms were adequate for processing data from earlier X-ray telescopes, D. A. van Dyk and H. Kang showed MCMC is more suited to processing the richer data available from the Chandra telescope, as explained in [61]. Chandra data is also open to alternative processing, in [7] M. Bonamente et al describe how MCMC can be used to combine datasets (Chandra X-ray data and Sunyaev-Zel'dovich effect data) to determine the distance to galaxy clusters in a manner more computationally efficient than earlier methods.

At the Institute for Robotics and Intelligent Systems, Los Angeles, Zhao and Nevatia developed a MCMC approach for segmenting individual humans in a high density scene (such as a crowd) acquired from a static camera [69]. Their technique allows for the partial occlusion of humans by other humans and has obvious uses for video surveillance and event inference. A MCMC solution for a similar application is covered (in considerably greater detail) by K. Smith at the EPFL, Switzerland in Chapter 1 of [54]. Smith's application places less emphasis on crowded images and feature occlusion, but was developed for analysing video feeds instead of static images. Whilst for the examples given in referenced papers runtimes are fast (~ 15 seconds per frame for Zhao's application and < 0.5 seconds for Smith's, though given the differences in circumstances and images, these times are not comparable) they involved only a small number of people. Parallelisation will help in coping with larger crowds, larger images and for taking less time per frame - for surveillance and CCTV footage analysis real-time processing is desirable.

There are numerous examples of MCMC in medical imaging applications. For instance, a MCMC algorithm for constructing a model for cells in an area of cartilage growth viewed using confocal microscopy was developed by F. Al-Awadhi (from Kuwait University), C. Jennison and M. Hurn (from the University of Bath) [2]. At the School of Mathematical Sciences, University of Nottingham and Department of Statistics, University of Leeds, I. Dryden, R. Farnoosh and C. Tayor, developed a method of segmenting images of muscle fibres using MCMC to describe them as Voronoi polygons [19].

Though this thesis places an emphasis on image processing applications, MCMC more often applies to non-image datasets. Whilst the parallelisation methods given in chapter 5 are predominantly restricted to datasets whose features are spatially localised, the methods presented in chapters 3 and 4 can be applied to any MCMC application. One significant example of a non-image application is its role in bioinformatics, in particular the construction of phylogenetic trees from nucleotide or amino acid sequences, as done by S. Li (at the Fred Hutchinson Research Center), D. Pearl and H. Doss (at the University of Ohio) [44]. A phylogenetic tree (or evolutionary tree) shows the evolutionary relationship between species or other entities that have a common ancestor. Bayesian inference through MCMC is an important way in which such trees can be constructed from available data, and a number of programs are available with which to perform this operation, including MrBayes [39] and BEAST [18]. MCMC also plays an essential role in statistics physics, lying at the heart of the tradition of simulation physics: understanding phase transition and other physical behaviour by constructing careful simulation experiments on the computer [42].

2.4 Existing Parallel MCMC

This section briefly describes existing methods for applying parallel processing to MCMC.

2.4.1 Multiple Chains

Despite the additional benefits and restrictions utilising Markov Chains brings, MCMC is still a Monte Carlo algorithm. As such, the typical Monte Carlo parallelisation method of using multiple chains on multiple computers (each with a separate random number generator and initial state) still applies. Obtaining many samples is embarrassingly parallel[*] as multiple chains can be run on multiple computers, each using a different initial model but keeping all other factors the same. Samples from all the chains can be simply grouped, not only reducing the time to obtain a fixed number of samples but also reducing the chances that all the sample will occur in local rather than global optima since the chains will be starting from different

[*]An embarrassingly parallel can be easily split and spread out over multiple processors, see section 2.1 for details.

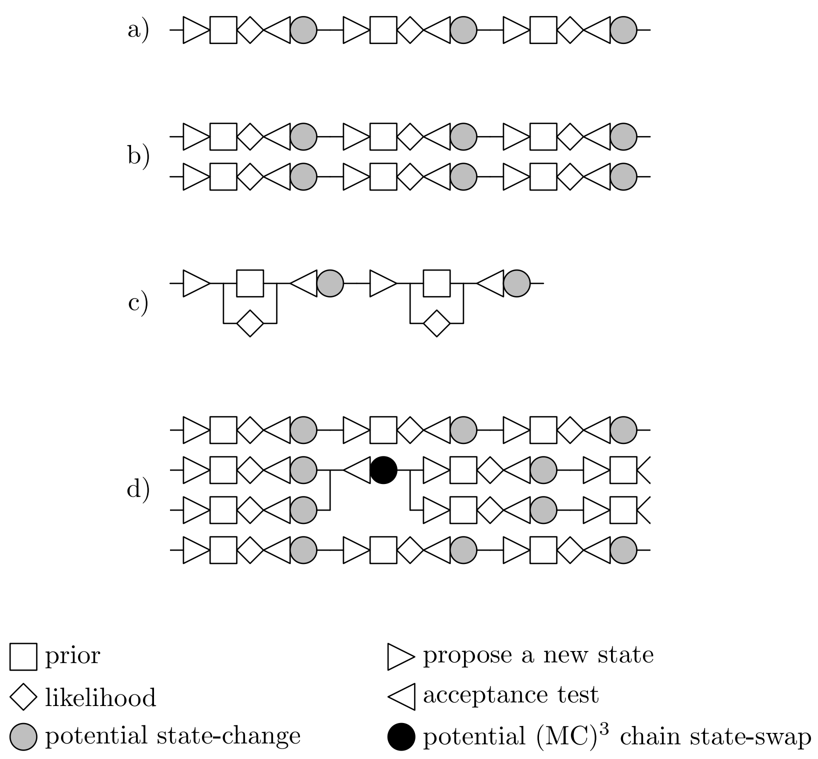


Figure 2.1: Comparison of existing parallel MCMC methods. Each row represents the sequence of actions performed by a single thread (read left to right, only the ordering is important). Vertical lines represent synchronisation between threads. a) Three iterations of normal MCMC. b) Multiple chains, using two threads (section 2.4.1). c) Intra-move parallelisation (section 2.4.2). d) Metropolis-Coupled MCMC, using four threads section 2.4.3)

positions in the state-space. This method is explained and considered in greater detail in [53] by J. Rosenthal from University of Toronto. As explained by Rosenthal the choice of the initial burn-in time is important but in general very difficult to select. Running multiple independent chains does not change the average necessary initial burn-in time for each individual chain (the time it takes for the chains to move from their initial states to achieving equilibrium around the states of optimal posterior probability), which for complicated and high-dimensional problems may be considerable.

Unfortunately the reason MCMC is used at all is that the statespace to be explored is generally extremely large. It may take a long time develop the chain's initial state into a state that suitably describes the input data. However, once the chain has converged on a solution, any alternative interpretations for the image data (alternative 'modes') are generally comparatively close in the statespace. For example, consider the vascular tree segmentation program developed by E. Thonnes et al.[57, 58] and implemented by D. Fan [21], both at the University of Warwick. Here a MCMC algorithm was constructed to map out the pattern of blood vessels seen in pictures of retina (the back of an eye). The model constructed here was that of a forest of binary trees. Constructing a graph describing the branching is a time-consuming undertaking (requiring several hours), but the alternative modes we would like to consider will differ only in a few localised places where the branching structure is less clear[*]. In such applications the vast majority of the processing time will be spent converging on the area of the statespace containing the potential solutions. Gathering sufficient samples to observe the possible solution modes need take only a fraction of the time since the distance between the modes in the statespace is comparatively minor. Though massive parallelisation by using many

[*]Should the modes differ to a major degree it is unlikely a single MCMC chain would deconstruct an existing well-matching forest to make visiting the alternative mode possible. In such circumstance alternative modes are best found by starting many chains (not necessarily simultaneously) with different initial states.

concurrently executing independent chains to gather samples is possible, it does little to address the long burn-in time. Initial convergence is the time-consuming factor and no matter how many chains are run the average time until convergence will remain unchanged.

2.4.2 Intra-move Parallelisation

Depending on how long individual iterations take to execute, there may be opportunities for parallelisation within each iteration. Since the likelihood and prior terms can be calculated independently, the processing can be done in parallel on separate processors, as shown in fig. 2.1 subfigure c). The value of doing this is dependant on the prior and likelihood calculations taking a significant proportion of the iteration's processing time and the prior and likelihood terms taking roughly equal time to execute. Under ideal conditions this will almost halve the time taken to evaluate a proposed move, although will do nothing to change the time required to construct the proposal in the first place.

For particularly large and complicated cases it may even be desirable to parallelise the actual calculations used to obtain the prior and/or likelihood term. All such terms are expressed as products[*] over the features comprising the chain's state, and as such are highly parallelisable (calculate the likelihood/prior contribution offered by each feature in parallel).

2.4.3 Metropolis Coupled Markov Chain Monte Carlo

The conventional approach to reducing the runtime of MCMC applications is to improve the rate of convergence so that fewer iterations are required. MCMC operates by performing a random walk through the statespace, preferentially favouring moves that shift the chain 'uphill' to the 'peaks' of high posterior probability density whilst avoiding the 'valleys' of low posterior probability density. Whilst a properly

[*]Though when coded will be implemented in the log domain, therefore as summations.

constructed Markov Chain will eventually explore the entire statespace, the probability of accepting a move down to a state of lower posterior probability will be low, thus many moves may need to be attempted before traversing a valley is achieved. To put this in context, 'crossing a valley' in the statespace involves performing a number of moves that when considered individually appear 'bad', but in the longer term allows new possibilities to be explored. For example, parts of a model may need to be deleted to allow new, potentially superior, interpretations for the data to be explored. In a 'craggy' statespace (containing many valleys and sub-optima peaks) a normal MCMC simulation will likely become temporarily trapped at local optima, the probability that the chain will descend a peak to possibly begin the accent of a different (hopefully more optimal peak) is sufficiently low that it takes a great many iterations to occur. Improving the ability of a chain to explore the statespace by crossing valleys is termed improving the mixing of the chain.

The simplest method of improving the mixing is to reduce the penalty incurred by travelling down statespace valleys through the addition of an exponent γ to the acceptance probability given by the Metropolis-Hastings transition kernel, i.e.

$$min\left[1, \left\{\frac{\text{prior}(R')}{\text{prior}(R)} \times \frac{\text{likelihood}(R'|I)}{\text{likelihood}(R|I)} \times \frac{p(R', R)}{p(R, R')} \times \mathcal{J}\right\}^{\gamma}\right] \tag{2.7}$$

By setting $\gamma < 1$ the probability that an arbitrary move will be accepted is increased, thus the number of moves (and time) required to explore the statespace is decreased. This is termed *heating* the chain. The disadvantage of such 'hot' chains is that they are less likely to stabilise on an optima (a 'peak' in the statespace). The exponent must be tuned such that a balance is found between allowing the Markov Chain to settle on areas of high probability, whilst allowing sufficient heat make the crossing of valleys in the statespace a realistic possibility.

C. J. Geyer at the University of Minnesota proposed a technique known as Metropolis-Coupled MCMC (termed $(MC)^3$) that improves mixing by using multiple MCMC chains with different stationary distributions [27]. One example, detailed in [28], uses a series of chains of increasing temperature (increasingly likely to accept arbitrary moves). One chain is considered ‘cold’ and configured as normal, the other chains are set to be at various higher temperatures. These ‘hot’ chains will be more likely to accept apparently unfavourable moves thus will explore the state-space faster than the cold chain. However, for the same reason they are less likely to remain at near-optimal solutions. Whilst samples are only ever taken from the cold chain, at intervals two chains are randomly chosen and their state’s swapped, subject to a modified Metropolis-Hastings test. This allows the cold chain to make the occasional large jump across the state-space whilst still converging on good solutions.

Whilst in its original incarnation $(\mathrm{MC})^3$ did not explicitly involve parallel processing, the work of G. Altekar and S. Dwarkadas at the University of Rochester [3], demonstrated that these MCMC chains can be efficiently performed in parallel, with each chain on a different processor (as shown in fig. 2.1 subfigure d), where the second and third chains are considered for being swapped). M. Harkness and P. Green at the University of Bristol have also shown that parallel $(\mathrm{MC})^3$ can also be applied in conjunction with delayed-rejection MCMC to further improve the convergence rate [36].

Altekar et al. proposed and implemented a parallel form of $(\mathrm{MC})^3$ applied to the problem of estimating phylogenetic trees using the parallel version of MrBayes [38]. Since state information for phylogenies can be several megabytes, a key aspect of their parallel $(\mathrm{MC})^3$ algorithm is to swap chain heats rather than chain states to keep communication costs minimal. Instead of each chain being assigned to a fixed processor/machine and the chain states being transferred between them, the evolving states remain on their host process and the ‘chains’ swap positions (the parameters governing the behaviour of a chain being far smaller in size compared to the chain’s state information). Near optimal speedups were demonstrated using both message passing and shared memory implementations on both large and small datasets.

$(MC)^3$ differs from our work in the manner by which parallelisation is used - $(MC)^3$ increases the mixing of the chain, improving the chances of discovering alternative solutions and helping avoid the simulation becoming stuck in local optima. Essentially it reduces the number of iterations required for the simulation to converge, whereas the new methods presented in this thesis (the subjects of chapters 3 to 5) reduce the time required to perform a number of iterations. The two approaches will complement each other, particularly since $(MC)^3$ requires only infrequent inter-chain synchronisation thus allowing its chains to be spread over multiple computers connected by a comparatively low speed interconnects, whilst the methods presented in chapters 3 and 4 of this document are best applied to a chain on an SMP machine. A cluster of dual or quad core/processor machines would be the ideal platform for such a setup.

2.4.4 Task Decomposition

In some applications it is possible to split the input dataset or identify traits that can be considered and processed independently, as in the case of subsequence-level parallelisation in the phylogenetic inference work done by X. Feng et al. at the University of South Carolina [22]. One of the parallelisation methods considered by Feng is the division of the data sequence amongst processors. As with $(MC)^3$ this method of parallelisation is coarse enough to work over a network (indeed, the two methods are used simultaneously in Feng's analysis), however it is very application specific. In the general case making such clean divisions in the input data or internal representation is not possible. For instance when processing images, naively bisecting the image and considering the two halves separately will lead to anomalies near the subimage boundary, potential imbalances in the degree to which each subimage converges, and a loss of the statistical principals underpinning the MCMC methodology.

2.5 Motivational Research

One of the more challenging applications of MCMC is image processing. Consider the task of identifying and describing an unknown number of features in an image. For instance, the counting of tree crowns from satellite images [51], tracking people in a crowd [70], mapping the paths of blood vessels in an image of a retina [58], identifying organs boundaries (such as the Thalamus or prostate gland) in Magnetic Resonance scales as sets of curves [20], or counting cells in a slide of a tissue sample (as in fig. 5.1). MCMC is well suited to such problems as the use of Bayesian inference permits prior knowledge to temper and guide the processing of the image data, and with reversible-jump MCMC allows for the uncertain dimensionality (the number of dimensions a model has may vary, for instance the number of features that may be found is not fixed but may change as the chain progresses). The construction of suitable prior and likelihood terms is often surprisingly uncomplicated, although the resulting simulation will require 'tuning' to efficiently, reliably and rapidly converge on an acceptable range of solutions. The main obstacle to the use of MCMC in such image processing applications is the long runtime required to conduct a random walk through the huge statespace that exists when dealing with non-trivially sized images containing any significant number of features. Image processing reversible-jump MCMC is therefore a suitable context in which to frame the analysis of MCMC parallelisation.

Consider a subset of the possible applications: counting tree crowns in from satellite images, tracking heads in a crowd, or counting stained cells in slides of a tissue sample can all be abstracted down to the task of recognising and counting independent circular artifacts (circles) in an image. By focusing our attention on parallelising this general case we avoid dealing with unnecessary application specific detail and can concentrate on testing the effectiveness of the various parallelisation methods. From the point of view of the parallel algorithms considered in this document the key characteristics are that the features are independent (not consisting of composite or interconnected structures, as the mapping of network of blood vessels from retinal scans would be [58]) and small compared to the size of the image; beyond that features may be as simple or as complicated as is required. Throughout this thesis circles are used as examples and in test applications for the sake of clarity and ease of understanding/testing*. Results using simple structures such as circles are also the most generally applicable, as to provide performance improvements the methods described in chapters 3 and 4 require each iteration to take a minimum time dependant on the hardware employed. Demonstrating the methods from those chapters work when using simple circles as features implies that performance benefits will also be gained when applied to more complicated features that take longer to process.

This section presents two Reversible-Jump MCMC algorithms for detecting circles in an image. Whilst a detailed understanding of these algorithms is not required for an appreciation of the MCMC parallelisation methods that are the focus of this thesis, the intention of this section is to provide an example context for the aforementioned parallelisation schemes and to support future work by providing a starting point from which more complicated image processing applications may be constructed. The implementation of these algorithms is used for testing all the parallelisation methods described in subsequent chapters, and the comments on optimisations for this implementation (section 2.6) may be of interest to potential MCMC implementers. Note that the circle identification algorithms presented here are not intended to directly compete with existing alternatives (i.e. genetic algorithms [6], Hough transform [68], or fast-finding-and-fitting [16]), but rather serve

*The parallel algorithms will work just as well if the features to track are polygons or irregularly shaped blobs with a complex internal structure. Such complicated features would require more elaborate expressions for the prior and likelihood, a larger set of potential moves to accommodate the extra variables used to describe each feature, and substantially more work to ensure that the simulation parameters such that the chain would converge promptly, all of which would be superfluous for this thesis on parallelisation methods.

to demonstrate how MCMC may be employed and parallelised.

2.5.1 Feature Boundary Recognition

Consider the task of identifying (potentially hollow) circular structures/features of a specified mean size in an image. Given a bitmap image I such as the top left image in fig. 2.2, we wish to produce a configuration (collection) of circles C where each circle c is represented by its radius c_r and the coordinates of its centre (c_x, c_y). We will use MCMC to shape an initial configuration of circles into suitable description for image I. The prior term for a configuration will be determined by how well it fits certain simple assumptions, such as the number, size, and distribution of circles through the image. The likelihood term involves comparing the image to the configuration, we will do this by identifying the probable circle boundaries in the image by the use of a standard edge detection algorithm, the calculated boundaries from the configuration of circles can then be compared with the actual boundaries found in the image.

To identify and describe the edges in the image we will use the edge detection method of Sobel filters [1]. First proposed in 1968, the Sobel filters determine the gradient (the rate of change) of the image intensity at every pixel in the image - they emphasise all the boundaries in an image. They operate as a pair, one for detecting horizontal edges and one for vertical edges. Each filter produces two values for each pixel in the input image: a magnitude value representing the rate of change of pixel intensity across that filter's direction, and an orientation value indicating the direction of that change of intensity. The results from the two filters are then combined to create two new images. The first is an 'edge magnitude map' (or just 'edge map') of the original image showing where the edges (pixels where pixel-intensity is rapidly changing) are, as shown in the top right image of fig. 2.2. The second is an edge orientation map, providing the direction of the greatest rate of change in intensity for each pixel (i.e. describes the direction of the edges in the

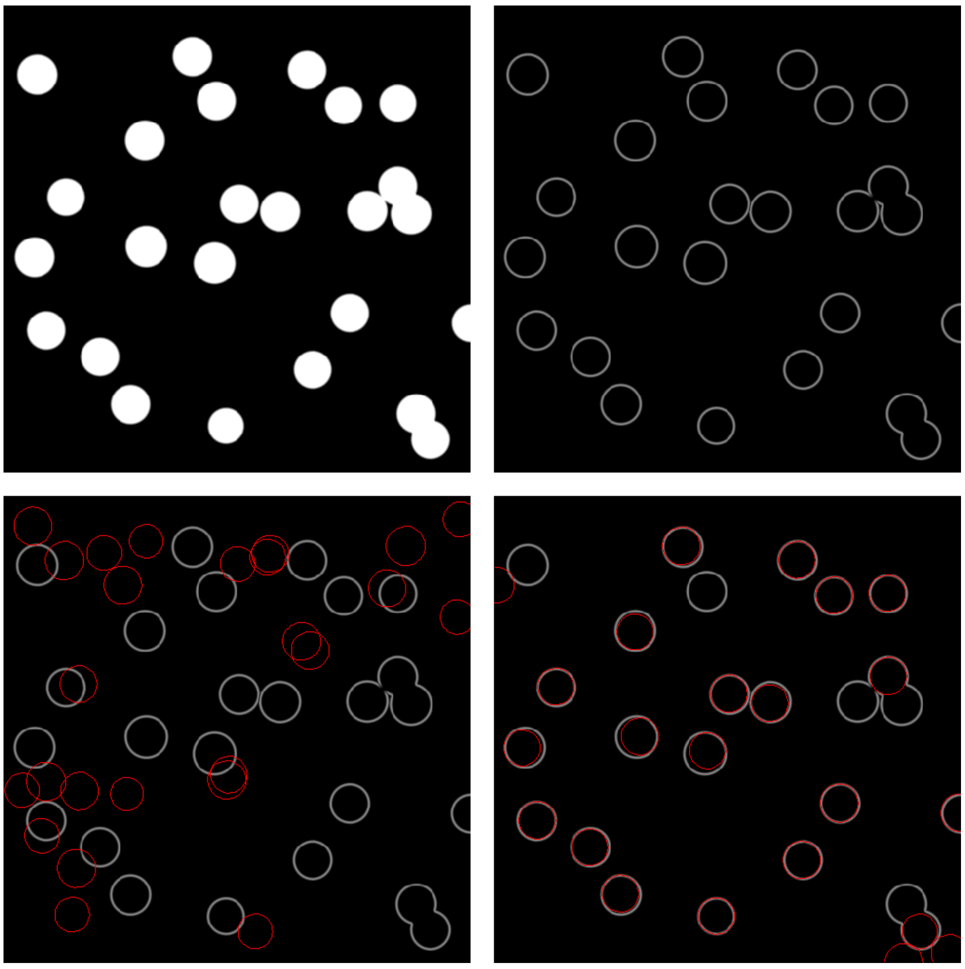


Figure 2.2: Demonstration of the circles recognition program. Top left: The initial image. Top right: The Sobel filtered image used by the MCMC simulation. Bottom left: The initial randomly generated configuration overlaid on the image. Bottom right: The configuration after 10 000 iterations (approx 4 seconds processing time).

edge map).

The first step is to add a small amount of random uniform noise to the image, to improve the stability for later steps in cases where there are large areas of little or no natural variation in the image. Next we enhance the salient aspect of the features to be identified by applying the Sobel filters to the image to extract directional edge information. If the pixel intensity[*] values for the image data are given by the function $T(x, y)$, then the standard Sobel filters from [1] can be expressed as follows:

$$G_h = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * T \tag{2.8}$$

$$G_v = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * T \tag{2.9}$$

where G_h is the horizontal component of the intensity gradient of each pixel in the image, G_v the vertical component of the intensity gradient, and '$*$' is the (two dimensional) convolution operator[40]. From these, the total magnitude of the intensity gradient for each pixel can be calculated as the length of the hypotenuse of a triangle of height $G_v(x, y)$ and length $G_h(x, y)$:

$$M_{\text{sobel}}(x, y) = \sqrt{G_h(x, y)^2 + G_v(x, y)^2} \tag{2.10}$$

Since we have the magnitude of both the horizontal and vertical components of the gradient, we can obtain the direction of the gradient through basic trigonometry. The edge orientation map (the direction of the intensity gradient for each pixel) is therefore

$$\Theta_{\text{sobel}}(x, y) = \tan^{-1} \frac{G_v(x, y)}{G_h(x, y)} \tag{2.11}$$

Note that this pre-processing takes negligible time compared to the MCMC simulation that follows.

[*]For colour images the average of the pixel's three colour values

A random configuration of circles is generated from the known *prior* knowledge (information we may assume we know about the image, obtained from heuristics and experience of similar images) and is used as the initial state of the Markov Chain. A MCMC simulation using the Metropolis-Hastings kernel (eq. (2.6)) is then applied to the configuration, magnitude and orientation maps to produce the desired circle configuration. As described in section 2.2.4 such a kernel requires three components: a prior term to evaluate what we 'expect' about the configuration's properties, a likelihood term to evaluate how the configuration fits with actual image, and a set of moves that may alter the configuration.

The Prior Term

The prior φ of a configuration C of circles $c^1..c^{|C|}$ will be constructed utilising assumed knowledge of the following (note that these values will estimate what we expect to find in the image, and need not be totally accurate):

- The number of circles in the image is $\approx \lambda$
- The mean circle radii is $\approx r_\mu$
- The standard deviation in circle radii is $\approx r_\sigma$
- The circles are uniformly distributed (were this not the case then an expected probability density map or formula could be referenced to take into account the distribution of circle locations)
- Overlapping circles are rare (we will assign an arbitrary value with which to penalise overlapping circles)

The expression for the prior term must assign a high value to models that meet these criteria, and a low value to those that do not. First, consider the prior probability density of a single circle c, of radius c_r centred at the coordinates (c_x, c_y). This will be constructed from terms evaluating the circle's radii, its position in the image, and its overlap with any other circle. Once we have a probability density for each circle we can construct the density for the complete configuration of circles, taking into account the number of circles we expect to find (λ).

The probability density function for a single circle will consist of three terms, one to rate its radius, one for its position and one for its overlap with other circles. For the radii term ϕ_r we need an expression that will reward circles with a radius close to r_μ. For this we shall use the probability density function of the normal distribution [64], with r_μ as its mean value, and standard deviation r_σ.

$$\phi_r(c) = \frac{1}{r_\sigma\sqrt{2\pi}} e^{-\frac{(c_r - r_\mu)^2}{2r_\sigma^2}} \tag{2.12}$$

This will return the maximum value when the radius $c_r = \lambda$ and lower values as c_r moves away from λ, as illustrated by fig. 2.3. The position term ϕ_p is trivial in this case as circles are assumed to be uniformly distributed throughout the image.

$$\phi_p(c) = 1 \tag{2.13}$$

Finally the term ϕ_o is used to penalise overlapping circles. Let V be the set of circles v_1 to $v_{|V|}$ that overlap with circle c, and $A(c, v)$ be the area of overlap between circles c and v which, from [63], is given by:

$$d = \sqrt{(c_x - v_x)^2 + (c_y - v_y)^2} \tag{2.14}$$

$$d_1 = \frac{d^2 - c_r^2 + v_r^2}{2d} \tag{2.15}$$

$$d_2 = \frac{d^2 + c_r^2 - v_r^2}{2d} \tag{2.16}$$

$$a(R, D) = R^2 \cos^{-1}\left(\frac{D}{R}\right) - D\sqrt{R^2 - D^2} \tag{2.17}$$

$$A(c, v) = a(v_r, d_1) + a(c_r, d_2) \tag{2.18}$$

To ensure that the prior term can be scaled to apply to both large and circles, ϕ_o should not scale directly with the area overlapping with other circles, but take into

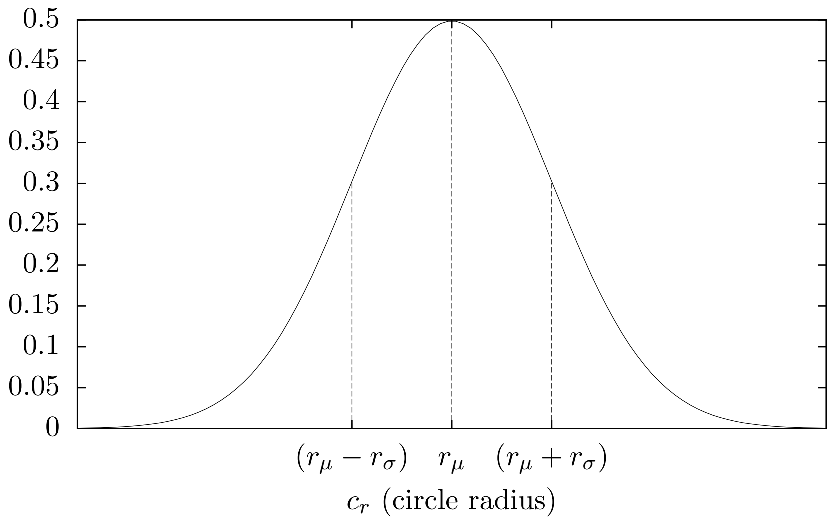


Figure 2.3: The normal distribution as it is used in ϕ_o. Given by eq. (2.12). In this example $r_\mu = 5$, $r_\sigma = 0.8$

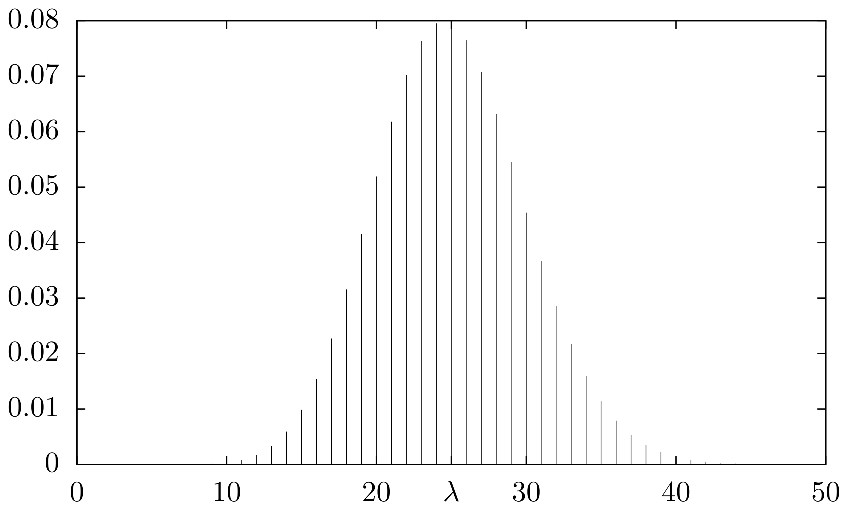


Figure 2.4: The Poisson distribution as it is used in the prior term (φ). Given by eq. (2.21). In this example $\lambda = 25$.

account the area of the circles with which it overlaps*:

$$\frac{\sum_{j=1}^{|V|} A(c, v_j)}{\pi \sum_{j=1}^{|V|} (v_{jr})^2} \tag{2.19}$$

(v_{jr} is the radius of the circle v_j). Additionally this term needs to evaluate to 1 when there are no overlapping circles, and provide an increasing penalty as the overlapping area increases. Both these issues can be satisfied by using eq. (2.19) as the exponent to a constant, such as e.

$$\phi_o(c) = e^{-\beta \left[\frac{\sum_{j=1}^{|V|} A(c, v_j)}{\pi \sum_{j=1}^{|V|} ((v_j)_r)^2} \right]} \tag{2.20}$$

β (> 0) is a control variable used to tune the magnitude of the ϕ_o term, thus the prior's tolerance for overlapping circles.

The probability density of a single circle will be given by $\phi_r \phi_p \phi_o$. Now we consider the probability density of the configuration as a unified entity. The only global property we need consider is the size of the configuration ($|C|$). We shall determine the probability density for this value by utilising a Poisson distribution [65]. If the expected number of circles is λ then the Poisson distribution gives probability that there are exactly $|C|$ circles in the model as

$$\frac{\lambda^{|C|} e^{-\lambda}}{|C|!} \tag{2.21}$$

which is plotted in fig. 2.4. Combining this with the product of the probability densities for each individual circle gives us the complete prior term:

$$\pi(C) = \frac{\lambda^{|C|} e^{-\lambda}}{|C|!} \prod_{i=1}^{|C|} \phi_r(c_i) \phi_o(c_i) \tag{2.22}$$

*The area of a circle of radius r is πr^2

The Likelihood Term

The likelihood assesses how good the image I is as an interpretation of the given configuration C of circles $c^1..c^{|C|}$. To do this we first identify the edges in the image I (the lines where the pixel colour/intensity is changing rapidly) using Sobel filters as described on page 43. This gives us two maps of the original image $I(x, y)$: an edge magnitude map $M(x, y)$ showing where the edges are, and a edge orientation map $\theta(x, y)$ giving the direction in which each pixel's intensity gradient is changing. With these we can determine if image has the 'correct' edge and orientation at each point where we 'know' that a circle should be[*].

The likelihood $L(c|I)$ of single circle c is obtained by taking samples from the edge and orientation maps at a number of sample points $p^1..p^K$ spaced at regular intervals on the circumference of c (sampling every pixel on the circumference will in most cases be unnecessary and prohibitively time consuming, K can be set so as to provide a good balance between efficiency and accuracy as determined by the needs of the application). For a point p at (p_x, p_y) on the edge of a circle centred at point (c_x, c_y), the orientation of that edge is given by

$$\tan^{-1} \frac{p_y - c_y}{p^i_x - c_x} \tag{2.23}$$

To compare this with the actual orientation of the corresponding pixel we take the difference of the two angles and take cosine of the result:

$$\cos\left(\theta(p^i_x, p^i_y) - \tan^{-1} \frac{p^i_y - c_y}{p^i_x - c_x}\right) \tag{2.24}$$

Orientations that match perfectly thus return a value of $\cos 0 = 1$. To get the likelihood of that point we take the product of the cosine of the differences in orientation and the edge magnitude at that point $(M(p_x, p_y))$, and likelihood of an entire circle c taken as being the product of the likelihood of every sample point

[*]recall that the likelihood is the probability of the image I *given* model M

taken around c:

$$L(c|I) = \frac{1}{K}\sum_{i=0}^{K} M(p_x^i, p_y^i).\cos\left(\theta(p_x^i, p_y^i) - \tan^{-1}\frac{p_y^i - c_y}{p_x^i - c_x}\right) \tag{2.25}$$

The number of sample points K is fixed for any one run of the program. Increasing the value of K is an easy way of increasing the workload per iteration, which is useful for testing the effectiveness of parallelisation methods in different conditions. For this reason the likelihood term is normalised with respect to K so K may be varied without altering the balance between the prior and likelihood terms in the Metropolis-Hastings test (see section 2.5.1 for more information).

The likelihood of the whole configuration is taken as the product of the likelihoods of each circle in the configuration:

$$L(C|I) = \prod_{c \epsilon C} L(c|I) \tag{2.26}$$

Recall that this cannot be an absolute probability, but it is proportional to the true probability (see section 2.2.4). Note that if the magnitude of each pixel sampled for a circle is 0 the likelihood contribution from that circle will be 0, as will the likelihood of any configuration including that circle. A circle likelihood of 0 must therefore be avoided, hence the addition of the minor random noise to the initial image. A circle likelihood would only be 0 if the edge magnitude of every pixel sampled was 0, the random noise ensures this will not be the case unless the circle is placed entirely out of the bounds of the image (which will be forbidden by the implementation of the available moves).

The Moves

If the total number of features to be found in an image is not known precisely, the potential moves with which to modify a model (in this case a configuration of circles) are birth, death, split, merge, alter position and alter radius. Birth moves insert a circle with uniform random coordinates and a radius sampled from the normal distribution using r_μ and r_σ. A death moves remove a random circle. A merge move removes two circles reasonably close together and averages their positions and radii to create a new circle. A split move does the opposite, replacing one circle with two circles from the prior distribution that if merged would yield the original circle. Alter radius and alter position moves select a random circle and choose a new position/radius from a normal distribution centred on the old value and with a preset variance. An alternative would be to resample the radius or position from the prior distribution, but in this case the radius/position is adjusted rather than re-sampled to give a higher acceptance rate to the alter moves, at the cost of slightly poorer mixing.

Tuning The Metropolis-Hastings Test

As covered in section 2.2.4 the probably of accepting a statechange from configuration C to C' is

$$\alpha(C, C') = \min\left[1, \frac{\varphi(C')L(C'|I)}{\varphi(C)L(C|I)} \frac{p(C', C)}{p(C, C')}\mathcal{J}\right] \tag{2.27}$$

To achieve a chain that does converge (and converges in a reasonable timescale) additional variables need to be added. Exponents ω_p and ω_l are applied to the prior and likelihood terms respectively to allow them to be balanced against each other. Without these ω terms either the likelihood or the prior would receive undue dominance in the results of the Metropolis-Hastings tests. A strongly dominant prior term results in random circle placement as the image data is not given enough weight. A strongly dominant likelihood leads to the creation of an excessive number of circles as the likelihood places no limiting factor on the number of circles in the model, potentially allowing an unlimited number of circles to be place on top of each other. Correct balancing between the prior and likelihood terms prevents these eventualities by favouring favouring models with close to the expected number of circles (λ), penalising circles that overlap (ϕ_o), whilst also favouring circles that match the image data ($L(C|I)$).

A general exponent γ is also needed to control the 'heat' of the chain - how generally accepting the test is. 'Heat' in this context is a concept taken from simulated annealing [43, 62], and refers to how easily the chain will shift states. 'Heating' a chain (by setting $\gamma < 1$) makes it more likely *any* arbitrary move will be accepted by the Metropolis-Hastings test, thus a 'hot' chain will be more likely to escape local optima, and more readily explore the statespace. 'Cooling' a chain (by setting $\gamma > 0$) decreases the likelihood that an arbitrary move will be accepted, making the chain's state more stable from iteration to iteration and less likely to backtrack into 'inferior' states, though this does increase the risk of becoming trapped in local optima. Whilst simulated annealing uses the gradual cooling as the means to drive the convergence to states with the maximum posterior probability, MCMC uses the fact that the Markov Chain will eventually converge to its stationary distribution as the means to find the maximum posterior probability. The heat term γ is used to set a single constant 'temperature' for the MCMC simulation, too 'cold' and the chain will rarely explore new states, whilst an excessively hot chain will not converge on a sufficiently detailed model*.

The final Metropolis-Hastings test used was

$$\alpha(C, C') = \min \left[1, \left\{ \left(\frac{\varphi(C')}{\varphi(C)} \right)^{\omega_p} \left(\frac{L(C'|I)}{L(C|I)} \right)^{\omega_l} \frac{p(C', C)}{p(C, C')} \mathcal{J} \right\}^{\gamma} \right] \tag{2.28}$$

Suitable values for γ, ω_p, ω_l, and β (the modifier controlling the tolerance for overlapping circles, see eq. (2.20)) were found by trial and error for each type of image to analyse (images with substantially different characteristics required different values). The rate of convergence of the MCMC chain may be improved by the fine tuning of these parameters (and the specifics of the prior and likelihood formula's), however determining how to obtain the optimum values for these variables (and in turn how to achieve the optimum convergence rate) is beyond the scope of this thesis. The parallelisation methods covered in chapters 3 to 5 will operate regardless

*Note that the Metropolis-Coupled Markov Chain Monte Carlo aka $(MC)^3$ uses multiple chains each set to a different temperature to improve the rate of convergence, see section 2.4.3

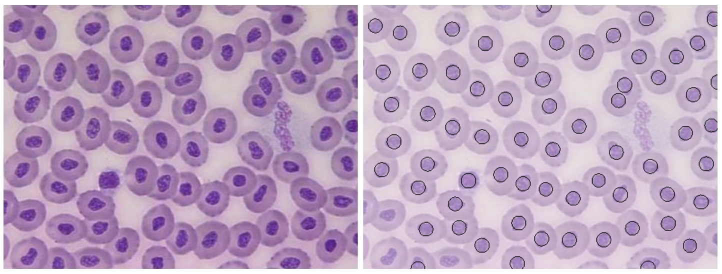


Figure 2.5: Demonstration of the pixel-intensity based feature recognition program. Left: the initial image. Right: the configuration after 10,000 iterations (approx 4 seconds processing time), having started from a random configuration.

of the rate of chain convergence.

2.5.2 Circle Intensity Recognition

The algorithm in section 2.5.1 is not necessarily the best approach to take, as the use of the Sobel filters to extract edge data that is then used as the basis for the likelihood term is not without its drawbacks. Though general purpose (in its ability to detect many types of circle) and quick to calculate the likelihood (sampling a relatively small number of pixels compared to the circle's size), the fact that we are searching for an edge (represented in the edge map as a thin line) poses difficulties. To obtain a strong likelihood signal a circle must be placed almost directly on target so there is little room for fine tuning by making a small move to incrementally nudge a circle to a more favourable position. Rapid convergence is more likely if we can impose some additional constraints on the original image. For example, consider the practical problem of identifying or counting dyed biological cells in a tissue sample, as in fig. 2.5. The easiest part of a cell to identify is its nucleus, as the nuclei are the only large solid blocks of intense colour in the image. Whilst the edge detection algorithm is serviceable, it would be better to search for blocks of high colour intensity directly, rather than the thin lines obtained from the Sobel filters (the blocks of colour intensity are a more reliable target than the thin lines in the edge map). Should the image also contain regions of high colour intensity that were *not* cell nuclei (a different shape say) the feature boundary detection method would be preferable.

The formula for the prior term and the moves described in the previous section can be reused for this new algorithm. The Sobel filters can be dropped in favour of a simple hew-balancing operation that emphasises the specific hew we expect the cell nuclei to be. For simplicity, in this case we will simply invert the image colours and take each pixel's average intensity. The likelihood contribution for each circle will be changed to be the average intensity of all the pixels enclosed inside that circle. Specifically, for a circle c with coordinates (c_x, c_y) and radius c_r, let $P(c)$ be the set

$$\{\forall(x,y)\epsilon(\mathbb{Z},\mathbb{Z}) : (x-c_x)^2 + (y-c_y)^2 \leq c_r^2\}$$

and $I(x, y)$ be the intensity (average of all colour values) of the pixel at (x, y).

$$L(c) = \frac{1}{|P(c)|} \sum_{(p_x,p_y)\epsilon P(c)} I(p_x, p_y) \tag{2.29}$$

2.6 Optimising the Implementation

In both algorithms presented above, the likelihood is defined solely in terms of each feature (considered in isolation) in the current chain's state. The likelihood of the chain is the product of the likelihood contributions from each of these features. The contribution from each feature is localised, the 'likelihood' of a single feature (circle) depends only on the value of pixels close to that feature yet is independent of the location of any other feature. It is therefore unnecessary to recalculate the likelihood from scratch using eq. (2.26) at each iteration. Instead the *change* in likelihood from the previous state can be determined by obtaining the likelihood contribution from all of the circles that are being added, removed or changed in that iteration. For a move that adds the circle c to a configuration of likelihood $\mathcal{L}$, the new likelihood $\mathcal{L}'$

is given by multiplying c's likelihood contribution to the existing likelihood:

$$\mathcal{L}' = \mathcal{L}.L(c) \tag{2.30}$$

The likelihood for moves that alters a circle in some way (be it its radius, position, or both) can be quickly calculated by viewing the alteration as the deletion of the original circle c and the simultaneous addition of a new circle c' (representing the 'altered' state).

$$\mathcal{L}' = \mathcal{L}.\frac{L(c')}{L(c)} \tag{2.31}$$

Similarly the net likelihood term for a move that merges circles c_a and c_b into a single circle c' is the same as removing circles c_a and c_b then adding circle c'

$$\mathcal{L}' = \mathcal{L}.\frac{L(c')}{L(c_a)L(c_b)} \tag{2.32}$$

and for the reverse operation, the splitting of c into c' and c'' is

$$\mathcal{L}' = \mathcal{L}.\frac{L(c')L(c'')}{L(c)} \tag{2.33}$$

It is possible to combine these formulae with the Metropolis-Hastings transition kernel directly and cancel out $\mathcal{L}$ altogether as was done in [21], though in our implementation we leave $\mathcal{L}$ as a distinct entity to simplify the coding of the parallelisation mechanisms discussed in following chapters.

Calculating the likelihood by determining how the likelihood used in the previous MCMC iteration will change in reaction to a proposed move turns the likelihood calculation from $O(n)$ to $O(1)$ (where n is the number of features in the configuration) at the cost of slowly accumulated rounding errors (as each successive likelihood value is derived from its predecessor) resulting in a drift of the likelihood value over many iterations. This can be rectified by periodically forcing a full recalculation of the likelihood. Since the likelihood calculations will typically be performed in the log domain for numerical stability*, the full recalculation of the

*For efficiency reasons, the Metropolis-Hastings test should also be applied in the log domain, the 'exp' operation is computationally expensive and its use should be avoided wherever possible.

likelihood need be performed so infrequently as to have negligible impact on the program's runtime. As an additional effect, the reduction in the number of memory accesses required by the likelihood may result in a reduction in the percentage of time taken in memory accesses compared to pure computation for each move. This can equate to an additional saving in runtime when many threads are in operation by reducing the load placed on the memory bottleneck.

In principal the prior term may also be calculated in the same manner, but for the examples considered in this chapter all moves potentially modify either 'global' properties in the prior term (such as the expected number of features in the image) and/or the contribution to the prior given by all other features due to the overlap penaliser term ϕ_o, see section 2.5.1. The prior is therefore recalculated at each iteration (a $O(n^2)$ operation (due to the overlap penaliser term ϕ_o testing each circle against every other circle to identify overlaps). Depending on the algorithm and the MCMC move employed, it may not always be necessary to recalculate the prior. Consider a variant of the circle intensity algorithm where the colour of the circles is also identified though a modified likelihood calculation, but no colour is 'preferred' or 'expected' on a global scale, so colour is not featured in the prior. In such cases, the prior term need not be re-evaluated in moves that only modify variables that are not involved in the global component of the prior term, in this situation an alteration in a circle's colour would not require a prior recalculation.

2.7 Hardware

The following systems have been used for testing:

- AMD Athlon 64 X2 4400+ (dual-core), Linux 2.6.22-2
- Intel Xeon Dual-Processor, Linux 2.6.9-55
- IBM xSeries 330 Dual-Pentium III Processor, Linux 2.6.24

- Intel Pentium-D (dual core), Linux 2.6.18-36
- Intel Core2 Quad Q6600 (2x dual-core dies) Linux 2.6.18-36
- 56 Itanium2 Processor SGI Altix, Linux 2.4-21-sgi306rp52[*]

Note that the terms 'dual core' and 'dual processor' are not interchangeable. In both cases two threads of execution may proceed simultaneously (as oppose to a single processor machine which simulates simultaneous execution by interleaving the instructions of the threads) however a 'dual processor' computer has two physically distinct yet connected CPUs, whereas a 'dual core' refers to a CPU processor in which most processing functionality is duplicated within the unit (i.e. on the same die/integrated circuit). As the the processing cores of a dual-core machine are located within the same CPU, they are capable of synchronising and communicating (through on-board cache) much faster than the (physically separated) CPUs within a dual-processor computer could. The 'Intel Core2 Quad Q6600' is not a genuine quad-core, but a hybrid, it is a dual processor computer where each processor is itself dual-core.

To ensure that the multithreading primates (mutex locks) were not, on their own, causing substantial overhead, the multithreaded applications were constructed so that they could be run in 'sequential' mode. In this mode they used only a single thread, and performed only sequential MCMC, but they also established mutex locks at the same points in the program cycle that the multithreaded application would do. This test was deemed necessary after it was found that initial tests on an dual-processor Opteron machine running an early 2.4 Linux kernel had the runtime double just by the addition of the mutex locking and unlocking operations. For all the systems used to generate results in this thesis, the addition of pthread locks increased average sequential runtimes by less than 2%.

[*]This computing facility was provided by the Centre for Scientific Computing of the University of Warwick with support from a Science Research Investment Fund grant

Chapter 3

Parallelisation by Speculative Moves

In the preceding chapter the theoretical principals of the Markov Chain Monte Carlo method was described and the algorithm presented. MCMC is implemented as an iterative simulation that conducts a random walk through a probability distribution to find configurations with the highest posterior probability. Using the Metropolis-Hastings method, at each iteration of the simulation a change in state is proposed that is then accepted with some probability. This acceptance probability is obtained by determining how a new state compares to the old in terms of what is expected about the target solution and how it compares with the actual data available. If these relative probabilities are set correctly (something that is surprising straightforward to do) the successive state changes form a Markov Chain who stationary distribution is approximately equal to the intended target solution - in other words taken across a very large number of state changes, the most frequently visited states will be those that best describe the input data. Most current methods for improving the performance of MCMC algorithms aim to improve the rate of convergence - the number of iterations required for the chain to reach equilibrium (to have reached those states very close to the target state).

Observe that whilst a Markov Chain must perform state changes in a strictly sequential order, those proposed state changes that are rejected have no impact on the final state of the simulation. The fastest progressing Markov Chain is one where all proposals are accepted, not through a relaxed transition kernel test but because all proposed state changes progress to a preferable state. Whilst such a situation is impossible (else why bother with the statistical framework at all), this chapter shows that it is possible to compress a Markov Chain such that only those moves that are accepted consume real time.

This is achieved by considering a 'batch' of possible state changes simultaneously but allowing at most one of those potential changes to be used to effect an actual state change. The rejected iterations in each batch thus take negligible real-time to consider. Depending on the size of each batch of state change proposals the runtime required to perform a number of iterations may be reduced to (but not below) that required to perform only those iterations that would have resulted in a state change. Using the section 2.5 applications, reductions in runtime of 35 and 55 percent were obtained on SMP machines using two and four processors respectively.

3.1 The MCMC Program Cycle

As explained in section 2.2.3, a MCMC program starts with some initial state which is then modified one small step at a time until it is a satisfactory description for the supplied data. For each iteration a modification (a move) is proposed to transition from the current state x to a new state x'. This move is considered using a transition kernel (the function α in fig. 3.1) to give a probability for accepting the move (α'). A random number generator $rng()$ is used to determine whether to accept the move to x' (with probability α'). A move that is accepted is applied to the simulation's current state, a move that is rejected is discarded leaving the simulation unaltered. The program cycle for this is shown in fig. 3.1. The creation of proposed moves and the criteria of the transition kernel are such that the simulation's state tends

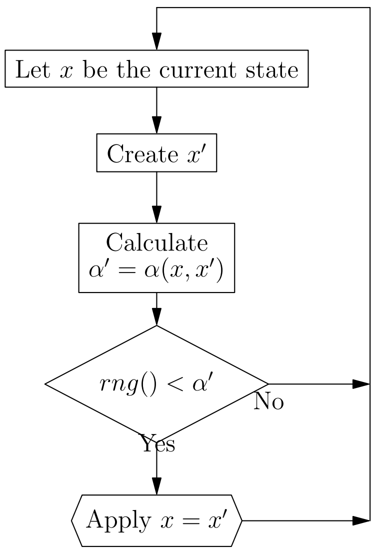

Let x be the current state
Create x'
Calculate $\alpha' = \alpha(x, x')$
$rng() < \alpha'$
No
Yes
Apply $x = x'$

Figure 3.1: Conventional Markov Chain Monte Carlo Program Cycle - one MCMC iteration is performed at each step of the cycle. $rng()$ is a function that returns a random number from a uniform distribution in the range 0 to 1.

towards a equilibrium distribution equal to the target distribution.

For this assertion on the eventual convergence of a simulation to hold, the statistical properties of the Markov Chain Monte Carlo algorithm must remain intact. Foremost is that the simulation is a Markov Chain, the 'next' state of the simulation must depend only on its present state and the fixed rules governing state progression. This appears to prohibit any form of parallel processing, as each new state must be derived solely from its predecessor. Fortunately it is possible to work around this rule and insert parallel processing that achieves substantial runtime reductions without invalidating the Markov Chain nature of the simulation using a method I have termed 'speculative moves'.

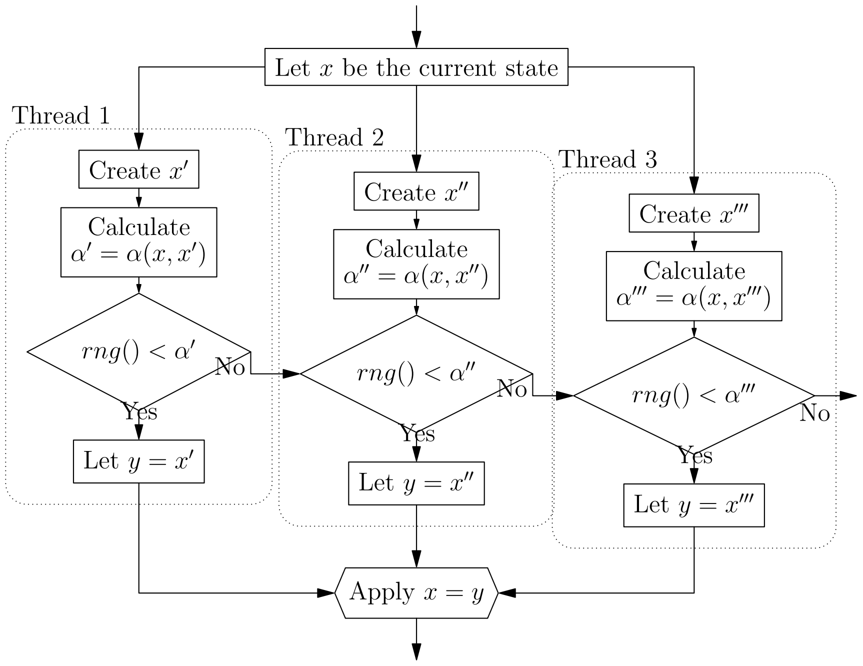


Figure 3.2: Speculative move enabled program cycle. In this case three potential moves are considered at each step of the program cycle. This translates to one, two or three MCMC iterations being performed, depending on whether the first and second potential moves are accepted or rejected.

3.2 Speculative Moves

Although by definition a Markov chain consists of a strictly sequential series of state changes, each MCMC iteration will not necessary result in a state change. In each iteration (see fig. 3.1) a state transition (move) is proposed but applied subject to the Metropolis-Hastings transition kernel. Moves that fail this test do not modify the chain's state so, with hindsight, need not have been evaluated. Consider a move to x'. It is not possible to determine whether x' will be accepted without evaluating its effect on the current state's posterior probability, but we can assume it will be rejected and consider a backup move to x'' in a separate thread of execution whilst waiting for x' to be evaluated (see fig. 3.2). If x' is accepted the backup move x''

- whether accepted or rejected - must be discarded as it was based upon a now supplanted chain state. If x' is rejected control will pass to the thread considering x'', saving much of the real-time spent considering x' had x' and x'' been evaluated sequentially. Of course, we may have as many concurrent threads as desired, so we may use x''' if x'' is rejected, then x'''', x''''', and so on. Obviously for there to be any reduction in runtime each thread must be executed on a separate processor or processor core. Interleaved threads will result in a net slowdown as the execution of speculative moves (that may or may not count towards the chain's iteration count) will delay the execution of 'normal' moves (that certainly will count).

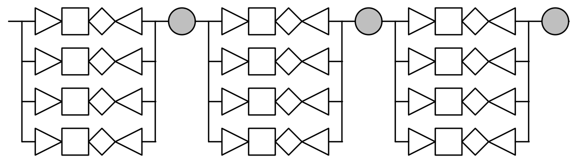
3.2.1 Comparison with Speculative Branching

The concept of speculative execution is already used to an extent in most modern processors. The use of pipelining (see section 2.1.5) means that instructions written by a programmer to be performed sequentially may be performed out of their intended order and potentially in parallel with one another (though with various safeguards to ensure these optimisations do not effect the end results). It is likely that a pipeline processor will reach a conditional branching instruction (an instruction that, depending on some value, may alter the flow of control of the program), such as an `IF` statement, before the value needed to decide which branch to take is available (i.e. that result may still be being calculated, later in the pipeline). Rather waiting (stalling) the pipeline until necessary results are available, it can be more efficient to speculatively continue processing one (or both) of the branches until the actual set of instructions to follow is determined. Once it is know which branch of instructions the conditional should take the speculatively executed instructions can either be confirmed as legitimate, or discarded, depending on whether the 'correct' branch was speculatively performed.

In both speculative branching and speculative moves, the speculative execution of code is used because of uncertainty over whether a certain set of actions need to be performed. Pre-emptively performing those actions then undoing or discarding them if it is later decided they should not be done can be more efficient that standing idle until the decision on those actions is made. Though similar, the two speculative methods are not identical. With speculative branching the speculative execution begins once a conditional branch has been reached, and only one of the branches will be the 'correct' path. In contrast, with speculative moves the speculative execution is performed in anticipation of a future branch point (the validity of the speculative move depending on whether the primary move will be rejected), and it is possible for multiple speculative moves to be considered simultaneously without any of them being discarded as 'wasted' computations*.

A related distinction is that the 'success rate' (the proportion the of speculative execution that does not get discarded) for speculative branching can be improved through heuristics and good record keeping. If a particular branch is visited multiple times in a program's run, branch prediction can be employed to guess which branch is the most like to be chosen, that is the branch that is then speculatively performed. Such a system is obviously not relevant to speculative moves.

There is also the obvious difference in parallel methodology (the many stages of a pipeline operating concurrently vs two or more distinct processors operating alongside each other) and in the time-scales between speculative branching and speculative moves. Speculative branching takes place inside a processor pipeline, and only lasts as long as it takes for the solution to the branch conditional to be produced. Speculative moves takes place over a much longer time-scale, the prior and/or likelihood calculations can be complex and involve a great many operations using all sorts of resources (different arithmetic operations, memory accesses etc.).



□ prior

◇ likelihood

● potential state-change

▷ propose a new state

◁ acceptance test

Figure 3.3: Speculative moves implemented using four threads. Each row represents a thread, vertical lines indicate synchronisation points between threads, shapes represent work being done. Time passes from left to right.

3.2.2 Implementing Speculative Moves

To be useful the speculative move must not compete with the initial move for processor cycles. In addition, the overhead for synchronising on the chain's current state, starting the speculative moves and obtaining the result must be small compared to the processing time of each move. An SMP architecture is most likely to meet these criteria, though a small cluster might be used if the average time to consider a move is long enough. As many speculative moves may be considered as there are processors/processing cores available, although there will be diminishing returns as the probability of accepting the mth speculative move is $(p_r)^{m-1}(1-p_r)$ where p_r is the probability of rejecting any one move proposal.

Figure fig. 3.3 shows how speculative moves would be applied on a quad-core system. Each row represents the sequence of actions performed by a single thread (read left to right). The vertical lines represent synchronisation points between threads, and the shapes represent work being done. The top row is the primary thread, the program's initial thread and the one performing the non-speculative

[*]Admittedly for this to be the case all bar the last move would need to fail their Metropolis-Hastings test, the MCMC chain would experience at most one state change.

move. The remaining rows show the threads that will perform speculative moves. Threads are not created or destroyed during MCMC processing, instead each of the threads on which speculative moves are performed is kept idle when not needed by waiting on a condition variable (see page 19). At the start of each iteration the speculative threads are signalled to begin work. Each thread (including the primary thread) then constructs and considers a considers a single move and determines the move's acceptance probability, reading the current state of the chain as stored in shared memory and storing its results in a thread-specific memory location. Each speculative move-performing thread performs a single speculative move then returns to its idle state to await the next iteration. The primary thread waits until all speculative moves have completed, then tests each in turn (reading each thread's results from shared memory and comparing the output of a random number generator to the thread's move's acceptance probability) until one move is accepted or all are rejected. Only when all other threads are idle may the chain's state be updated, if one of the moves has been accepted.

Speculative moves effectively compresses the time it takes to perform a number of iterations (see fig. 3.4), without changing the results of those iterations. The method will therefore complement existing parallelisation that involves multiple chains to improve mixing or the rate of convergence (such as $(MC)^3$ or simply starting multiple chains with different initial models), provided sufficient processors are available. As the other parallelisation methods have significantly fewer synchronisation points than speculative moves (speculative moves synchronise at the end of every step of the program cycle, $(\mathrm{MC})^3$, many chains etc. all synchronise after long periods of independent running) it is feasible for physically distinct computers to work on different chains, whilst each chain makes use of multiple cores/processors on its host computer for speculative moves.

When using sequential MCMC there are several methods for implementing the proposal and testing of potential new chain-states, such as working on the actual

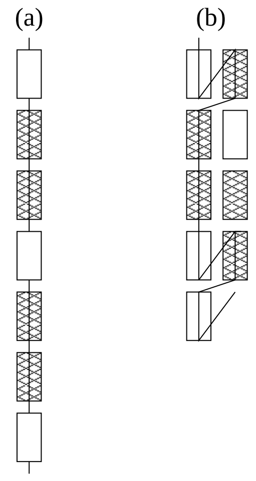


Figure 3.4: How speculative moves compress iterations into a smaller time period. (a) is normal MCMC, (b) uses speculative moves. The shaded blocks represent accepted moves, the white blocks rejected ones. The line indicates the moves in the order they are 'seen' by the MCMC algorithm.

chain's state then rolling back the changes made if the move is rejected. When using speculative moves the procedure for proposing and testing moves needs to operate without changing the original datastructure (until the move has been accepted, at least). Cloning the original chain state, making modifications, the calculating the prior and likelihood terms for this modified state is one possibility, but prohibitively expensive. What is needed are expressions for calculating what the prior and likelihood terms of a state will be after the application of a proposed change, without actually making that change to the datastructure. Implementing the prior and likelihood calculations in this way means that speculative moves will be applicable even when dealing with very large states (i.e. megabytes in size as in the case of phylogenies [3]), as the base state will reside in shared memory, and only the specific changes that will be made to that state will be stored and analysed on the threads performing the speculative computation.

3.3 Theoretical Gains

When using the speculative move mechanism with n moves considered simultaneously, each step of the program cycle (fig. 3.2) considers n distinct moves from the current state. The moves are considered in sequence, once one move has been accepted all subsequent moves considered in that step must be discounted (as they would not have taken place in a normal sequential implementation). Each step of the speculative move program cycle therefore performs the equivalent of between 1 and n 'conventional' MCMC iterations, depending upon which (if any) of the speculative moves was accepted.

What is the relationship between the number of steps of the speculative moves program cycle performed and the number of conventional MCMC iterations that occur? We will start by considering the different possible outcomes for each step. Let S_n be the number of steps performed by a speculative move MCMC program considering n moves per step, in which case S_1 is simply the sequential implementation of MCMC. Let S_R be the number of step that are rejected and S_A the number of step that are accepted, in which case

$$S_1 = S_A + S_R \tag{3.1}$$

When we have two moves considered in parallel at each step there are four possibilities: let S_{RA} be the number of steps where the first move was rejected and the second accepted, S_{RR} the number of steps where both move proposals were rejected etc. Continuing in this manner for S_3 gives us

$$\begin{aligned}
S_2 =& S_{RR} + S_{RA} + S_{AR} + S_{AA} \\
S_3 =& S_{RRR} + S_{RRA} + S_{RAR} + S_{ARR} + S_{RAA} + S_{ARA} + S_{AAR} + S_{AAA} \\
&\ldots
\end{aligned}$$

The number of iterations N_n performed by S_n steps is counted by summing the number of move proposals considered up to and including the first accepted move in each step. S_{RA} steps therefore counts for two iterations, whilst S_{AR} counts as one. The number of iterations performed by a specified number of steps of each type is therefore

$$\begin{aligned}
N_1 =& S_R + S_A \\
N_2 =& 2S_{RR} + 2S_{RA} + S_{AR} + S_{AA} \\
N_3 =& 3S_{RRR} + 3S_{RRA} + 2S_{RAR} + 2S_{RAA} + S_{ARR} + S_{ARA} + S_{AAR} + S_{AAA} \\
&\dots
\end{aligned}$$

Assuming the probability for rejecting any one move proposal is constant at p_r and substituting this probability in gives us:

$$\begin{aligned}
N_1 =& S_1(p_r + (1 - p_r)) = S_1 \\
N_2 =& S_2(2p_r^2 + 2p_r(1 - p_r) + (1 - p_r)p_r + (1 - p_r)^2) \\
=& S_2(p_r + 1) \\
N_3 =& S_3(p_r^2 + p_r + 1) \\
&\dots
\end{aligned}$$

Which, rearranging and expressing in terms of a fixed N gives:

$$\begin{aligned}
S_1 =& N \\
S_2 =& \frac{N}{p_r + 1} \\
S_3 =& \frac{N}{p_r^2 + p_r + 1} \\
&\dots
\end{aligned}$$

More generally, given that the average probability of a single arbitrary move being rejected is p_r, the probability of the i^{th} move in a step being accepted whilst all preceding moves are rejected is $p_r^{i-1}(1 - p_r)$. Such a step counts for i iterations. Including the case where all moves in a step are rejected (occurring with probability

p_r^n, counting for n iterations), the number of iterations (N) performed by S_n steps (where n is the number of moves considered in each step) can be expressed as

$$N = S_n\left[\sum_{i=1}^{n} i p_r^{i-1}(1-p_r) + n p_r^n\right] \tag{3.2}$$

$$N = S_n\left[\sum_{i=1}^{n} i p_r^{i-1} - \left(\sum_{i=1}^{n} i p_r^{i} - n p_r^n\right)\right] \qquad \text{rearrange} \tag{3.3}$$

$$N = S_n\left[\sum_{i=1}^{n-1}(i+1) p_r^{i} + p_r^0 - \sum_{i=1}^{n-1} i p_r^{i}\right] \qquad \text{by } \sum_{a=1}^{b} a x^a - b x^b = \sum_{a=1}^{b-1} a x^a \tag{3.4}$$

$$N = S_n\left[\sum_{i=1}^{n-1} p_r^{i} + 1\right] \qquad \text{simplify} \tag{3.5}$$

$$N = S_n\left[\frac{p_r - p_r^n}{1-p_r} + 1\right] \qquad \text{by } \sum_{i=a}^{b-1} = \frac{x^a - x^b}{1-x} \tag{3.6}$$

$$N = S_n\frac{1-p_r^n}{1-p_r} \qquad \text{simplify} \tag{3.7}$$

Rearranging for S_n

$$S_n = N\frac{1-p_r}{1-p_r^n} \tag{3.8}$$

which is plotted in fig. 3.5 for varying p_r. Assuming the time taken to apply an accepted move and the overhead imposed by multithreading are both negligible compared to the time required for move calculations, and that each iteration takes a constant realtime duration to perform, the time per step $\approx$ time per iteration. Therefore fig. 3.5 also shows the limits of how the runtime could potentially be reduced. For example, if 25% of moves in an MCMC simulation are accepted ($p_r = 0.75$), 100 sequential iterations are equivalent to ≈ 57 steps for a two-threaded speculative move implementation or ≈ 37 steps on a four-threaded implementation. Four thread speculative moves could therefore at best reduce the runtime of a MCMC application accepting 25% of its moves by about 63%, while the two threaded version could achieve up to a 43% reduction.

In practice speedups of this order will not be achieved. Threads will not receive constant utilisation (as they are synchronised twice for each iteration) so may

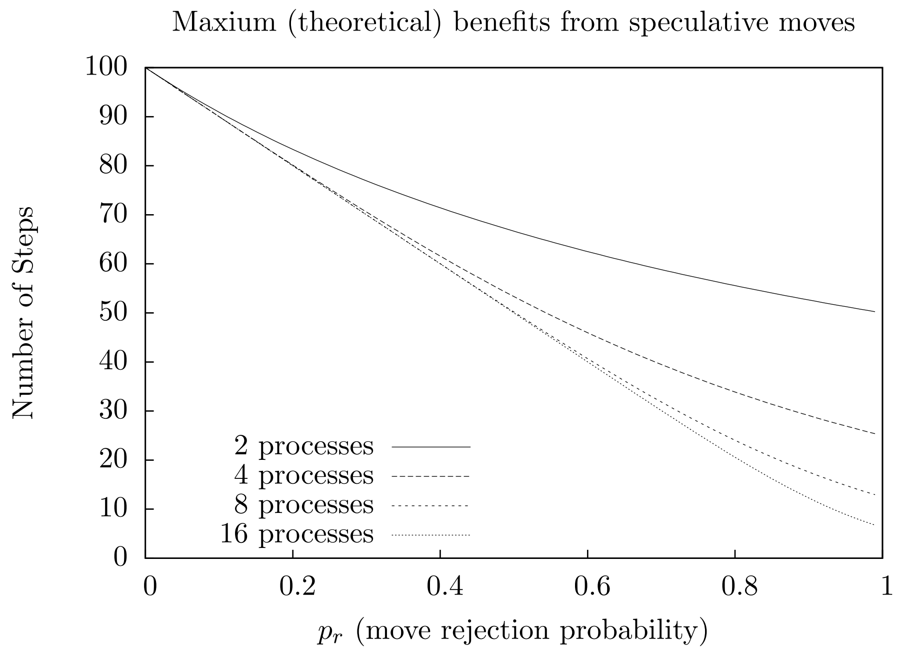


Figure 3.5: The number of speculative move 'steps' required to perform 100 iterations using multiple processors. The serial implementation performs exactly one iteration in each step, the number of steps will always be 100 irrespective of $\boldsymbol{p_r}$.

not be consistently scheduled on separate processors by the operating system. For rapidly executing iterations the overhead in locking/unlocking mutexes and waiting for other threads may even cause a net increase in runtimes. In addition, proposing and considering the moves may cause conflicts over shared resources, particularly if the image data cannot fit entirely into cache. Figure 3.5 can only be used to estimate the maximum possible speedup, actual improvements will fall short of this by an amount determined by the hardware and characteristics of the MCMC simulation to which speculative moves are applied.

3.4 Testing

For simplicity, and to allow a wide selection of input images to be used, we demonstrate our findings using randomly generated test images. These are randomly positioned white circles on a black background, with no other objects in the image. The circles were generated with the parameters (number, radii mean and variance) used by the prior calculations, with a check to avoid excessive overlapping of circles. A slight Gaussian blur was added to the image to make the circles easier to locate. More complex image processing examples have been studied, the reader is referred to [21, 45, 51, 58] as the applications per se are not the main focus of this thesis.

For each of the following tests a large, fixed number of iterations was performed (typically 10,000). Since the program execution time may vary due to the random nature of the MCMC method, variations in input images and background processes running on the test machines, each runtime value used in this thesis is actually an average taken over no less than 20 runs of the program[*]. Each run processed a different randomly generated image, using a different initial model and random number generator seeds. Since the effective MCMC algorithm in use has not been modified, the same resultant models will be produced after a fixed number of iterations irrespective of how many threads/speculative moves are used[†]. The traditional difficulty of determining when a MCMC program has 'converged' or completed its processing can therefore be ignored for the purposes of judging the speculative move parallelisation method. Likewise the efficiency of the circle-finding algorithm and the fine tuning of its various parameters is not relevant beyond the

[*]A legitimate number of repetitions as actual variation in runtime was minimal. With test runs typically taking tens of seconds to complete, short-term temporary delays/interruptions caused by background processes etc. would not cause any significant distortion of the results.

[†]The sole exception being if a constant value is used to seed the random number generator(s), in which case the presence of parallel processing may interleave access to a random number generator. This exception is not relevant, as all random number generators are provided with a unique seed based on the current time/date.

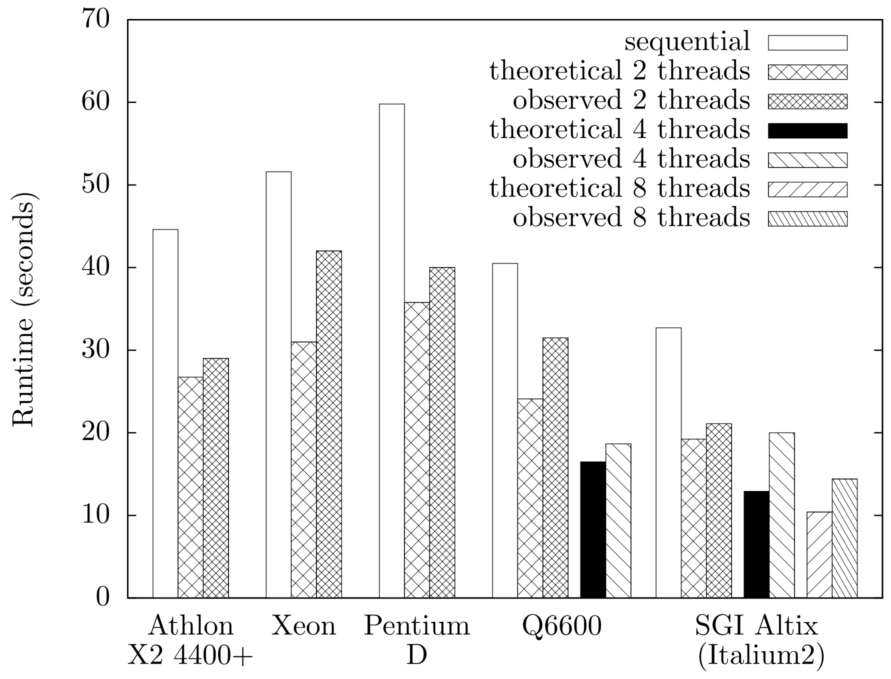


Figure 3.6: Speculative moves on different architectures, $\boldsymbol{p_r} \approx \mathbf{0.65}$

program's ability to maintain a stable feature count throughout its execution.

3.5 Results

Figure 3.6 show a comparison of runtimes across a number of hardware systems for one set of tests (other tests carried out with different parameters provided similar results.In this case the average move rejection rate was approximately 65%[*]. Some systems made more efficient use of the speculative moves method than others (due to differing overheads) but in all cases the use of speculative moves reduced the runtime to between 45 and 80% of that of the single threaded implementation.

Next we consider the effects of varying the time taken to perform each it-

[*]In practical applications the move rejection rate is not fixed by the developer, but may vary as the Markov Chain progresses. A detailed examination of this change in rejection rate and the effect it has on the runtime is left for future research.

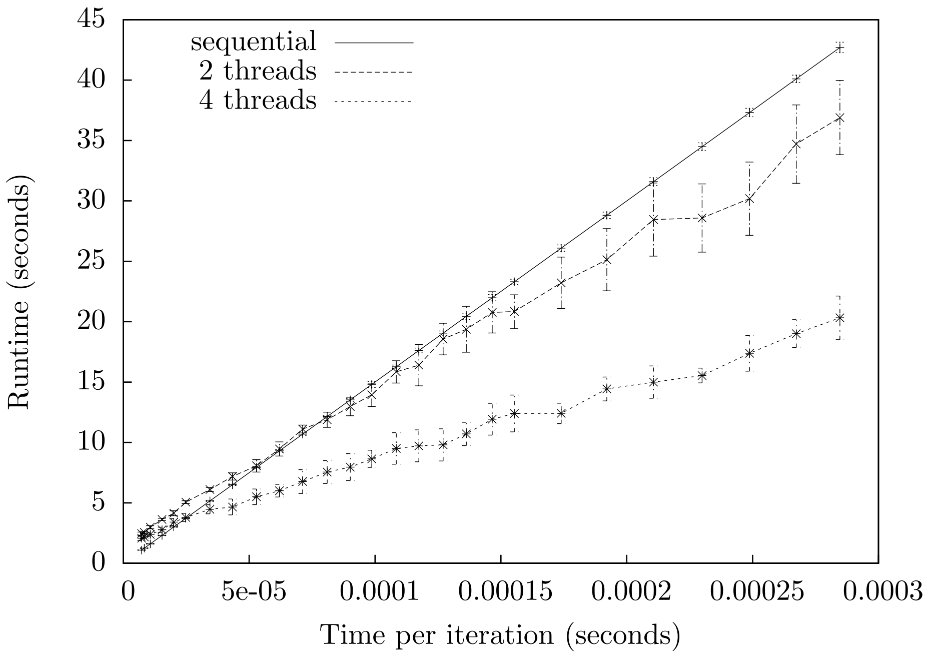


Figure 3.7: Runtime plotted against iteration time on the Q6600 (2x dual core). Simulation parameters set so as to demonstrate the point where speculative moves becomes beneficial. $\boldsymbol{p_r \approx 0.75}$

eration (obtained from the runtime of a program using only sequential execution). Two methods of varying the time-per-iteration can be used. The number of points sampled around each circle when performing likelihood calculations sample points can be increased so that the likelihood calculations for each circle take longer and involve more memory accesses. Alternatively the number of circles in the image can be increased, making the prior term take longer to process (most moves consider only the *change* they have on the likelihood, whereas the prior term must be recalculated in $O(n^2)$ for each move).

Figure 3.7 shows the runtimes using one, two and four threads on the quad core Q6600. Unlike the real-world task set in fig. 3.6, the parameters for the experiment show in fig. 3.7 were intentionally set to better determine the effect the

	# Threads	Iteration Time (μs)	Iteration Rate (s^{-1})
Xeon Dual-Processor	2	70	14 285
Pentium-D (dual core)	2	55	18 181
Q6600 (2x dual core)	2	75	13 333
Q6600 (2x dual core)	4	25	40 000

Table 3.1: Breakeven point when $\boldsymbol{p_r = 0.75}$

	# Threads	Iteration Time (μs)	Iteration Rate (s^{-1})
Xeon Dual-Processor	2	80	12 500
Pentium-D (dual core)	2	70	14 285
Q6600 (2x dual core)	2	130	7 692
Q6600 (2x dual core)	4	30	33 333

Table 3.2: Breakeven point when $\boldsymbol{p_r = 0.60}$

time spent on each iteration has on the benefits of speculative moves. Amongst other changes the per-iteration duration was varied by increasing the workload of the likelihood calculations, whilst number of circles in the model was kept constant at 15 so that the time-per-iteration remained steady throughout the simulation. For fast iterations the overhead involved in implementing speculative moves outweighs the benefits from the parallelisation. The points where the lines cross the 1-thread line represent how long each iteration must be before moves can be expected to start providing a real benefit (the point where the use of speculative moves ‘breaks even’, with the saving from considering moves simultaneously equalling the overhead required to implement that). These values are recorded for a number of alternative architectures in table 3.1 and table 3.2.

As a point of reference, the circles program searching for 300 circles using a modest 32 sample points performed around 2000 iterations per second ($500\mu s$ per iteration), whilst the vascular tree finding program from [21, 58] was generally performing $20 - 200$ iterations per second ($5 - 50ms$ an iteration). The tree crown

	# threads		
Machine	2	4	8
Xeon Dual-Processor	53	-	-
Pentium-D (dual core)	63	-	-
Athlon X2 (dual core)	75	-	-
Q6600 (2x dual core)	39	78	-
Altix (56 processor)	76	50	57

Table 3.3: The percentage of the theoretical reduction in runtime that was achieved for a set of experiments where $\boldsymbol{p_r \approx 0.78}$. Machines with higher values in the table are making more efficient use of their multiple processors.

finding program in [51] performed somewhere between ten to fifteen thousand iterations a second for small (200x140) images, processing larger images would be slower. We have found that many non-trivial MCMC applications will be well below the above iterations per second values and can therefore expect significant real-time savings by using speculative moves for real applications.

To determine the accuracy of the theoretical speedups as the move rejection rate is varied, the program was modified to ignore the calculated Metropolis-Hastings ratio and accept or reject moves based on a fixed pre-supplied probability. Moves that added or removed features were disabled for this test, otherwise the uniformly random acceptance of moves would cause the number of features in model to go to extremes (thousands of features, or only a few) and distort the runtimes. By fixing the model size each iteration will be sure to perform a 'normal' workload. The results for several machines are plotted in figs. 3.8 and 3.9. The results for the Pentium D are a good match for the theoretical results given that the theoretical values assume ideal (and unachievable) conditions. The Q6600 results are more mixed, whilst using four threads yields results reasonably close to the theoretical bound, when using only two threads the results are substantially poorer.

This difference between architectures is further explored in table 3.3, where the percentage of the maximum runtime reduction is displayed for the different

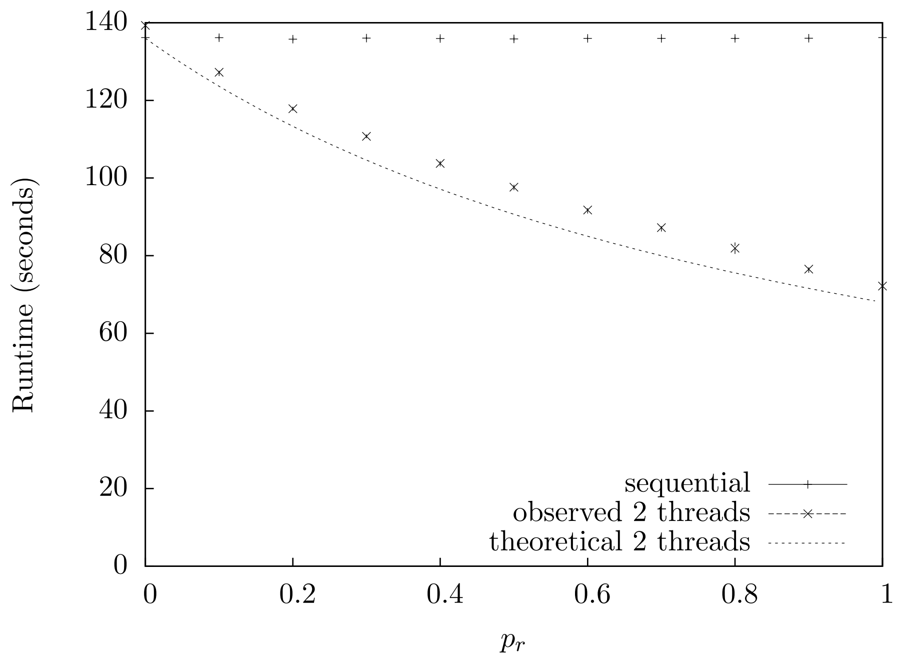


Figure 3.8: Runtime plotted against move rejection probability ($\boldsymbol{p_r}$) on the Pentium-D

architectures*. In the tests used to create this table p_r was ~ 0.78.

Considering all these results, the dual core machines (Pentium D and Athlon) gain more ($\sim$ 10%) from speculative moves than the dual processor Xeon due to the increased overheads involved in communications between the Xeon's two processors. Compared to the Pentium-D the Athlon X2 achieves roughly 10% more of the potential out of speculative moves. This is due to the differences in Intel and AMD's dual core designs, a detailed analysis of which is beyond the scope of this thesis. For the Q6600 using only two threads (thus two cores) the breakeven point is comparable to the dual processor Xeon, yet when using all four cores the

*For example, consider a sequential program that takes 100s to run. If the theoretical maximum benefit from speculative moves would reduce that to 65s, yet experimental results showed the program ran in 75s, the proportion of the maximum runtime reduction would be $\frac{100-75}{100-65} = \frac{25}{35} = \frac{5}{7} = 0.714...$ thus the table would show 71%

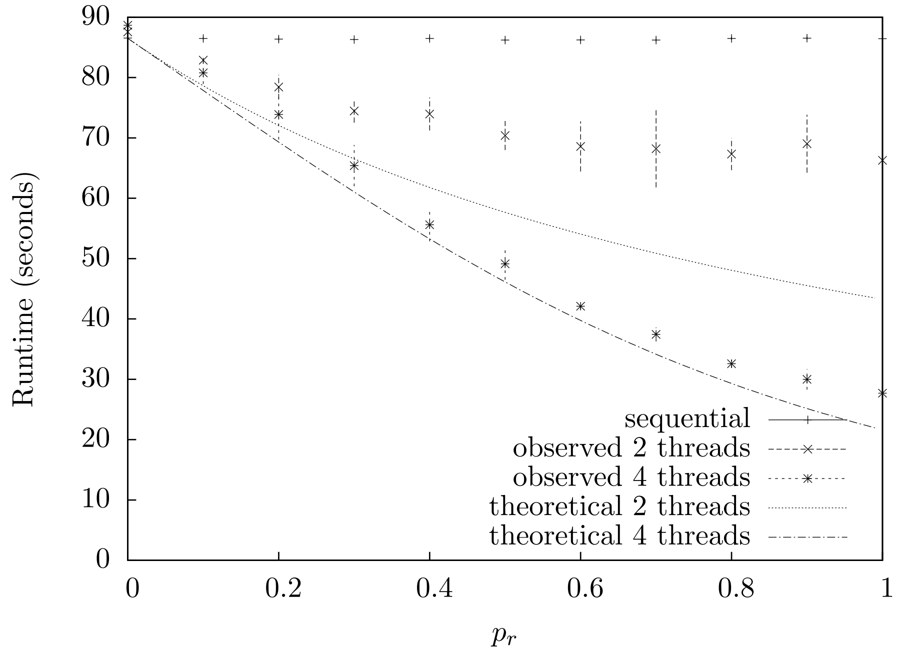


Figure 3.9: Runtime plotted against move rejection probability ($\boldsymbol{p_r}$) on the Q6600

breakeven point and fulfilment of speculative move's potential was the best of those machines examined ($25\mu s$ per iteration and 78% respectively). The difference in results between using two and all four cores of the Q6600 is due to the Q6600's scheduler allocating the threads on to alternate dies (the Q6600 has two dual-core dies), a sensible strategy when each thread belongs to a different program but in this case counterproductive as frequent synchronisation between the threads is required for speculative moves (synchronisation occurs at the end of every step through the program cycle, see fig. 3.3). Whilst all threads are operating on the same die the local on-die cache may be used, but when on separate processors the threads must communicate through the slower shared memory. In addition, when our MCMC program is using only two threads/cores, low priority processes would be scheduled on the two unutilised cores using up some of the shared non-processor resources that would otherwise have been used by the MCMC simulation (such as main memory). When all four cores were used such low priority processes were not getting as much processor time and so making little use of main memory (or the shared processor cache), allowing the greater performance benefits from the speculative moves. Conversely, the Altix achieved most of the potential speedup when only two threads were used (76%), but could only achieve 50-60% of the potential speedup when using more threads. The greatest reduction in runtime was achieved by the Altix using 8 threads (as in fig. 3.6), but this was not done as efficiently as in other scenarios. The difference in efficiency for the Altix is due to the arrangement of its 56 Itanium 2 processors: its processors are arranged in pairs, each pair having its own local cache. Information transfers between more than two threads must go via main memory (since the threads do not all shared the same local cache) and are therefore far less efficient.

3.6 Speculative Moves vs Intra-move Parallelisation

In section 2.4.2 it was suggested that the prior and likelihood calculations be conducted in parallel, should they take approximately equal time to process and be a significant proportion of the time-per-iteration. If this be the case, how does this method compare to the use of speculative moves?

Assuming that the processing time for the prior and likelihood terms are equal the potential benefit of performing intra-move parallelisation (when it is applicable) is slightly greater than that of performing a single speculative move each iteration. The presence of a single speculative move while $p_r = 0.75$ will typically reduce the runtime by about 40%, whereas under optimum conditions (fig. 3.10a) performing the prior and likelihood in parallel will reduce runtime of the prior/likelihood portion of the iteration by 50%. Unlike speculative moves intra-move parallelisation does not parallelise the work of proposing new states or the conduction of the acceptance test, but these are typically not expensive operations compared to the prior and likelihood term calculation.

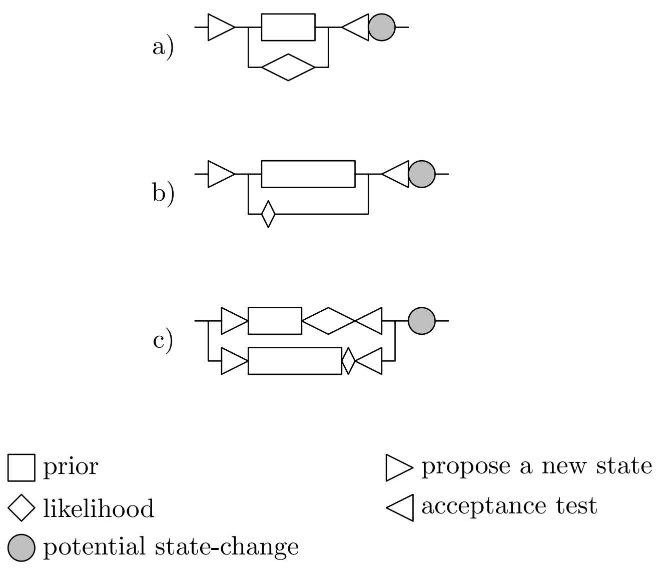


Figure 3.10: Comparison between speculative moves and intra-move parallelisation. Each row represents a thread, vertical lines indicate synchronisation points between threads, shapes represent work being done. Time passes from left to right. a) Intra-move parallelisation at optimum efficiency. b) Intra-move parallelisation with substantial difference between prior and likelihood processing times. c) Speculative moves.

There will then be some situations where intra-move parallelisation outperforms speculative moves. However, in practical applications the likelihood and prior terms are unlikely to take equal time to calculate thus intra-move parallelisation may not be as competitive as first appears (as shown in fig. 3.10b). For example, the case studies in chapter 2.5 are not suitable for this intra-move parallelisation as the likelihood calculation has already been reduced to an $O(1)$ order operation compared to the prior's $O(n^2)$ (achieved by taking advantage of the localised nature of some of the potential moves, see section 2.6).The prior term is by far the most computationally expensive operation, thus intra-move parallelisation would provide little benefit. In comparison, speculative moves is unaffected by the relative processing times of the prior and likelihood terms (fig. 3.10c). Additionally, intra-move parallelisation does not scale above the use of two processors* whereas speculative moves provides clear and predictable benefits for the use of four or more processors.

3.7 Summary

This chapter has shown how it is possible to consider multiple Markov Chain Monte Carlo iterations in parallel without violating the definition of a Markov Chain. Iterations of the MCMC program do not always result in a state change, and those iterations that do not cause a change in state can overlap without consequence. Since it cannot be determined in advance which iterations are state-changing we presume (speculate) that none of them are and consider multiple iterations in parallel. When an iteration is found that does changes the simulation's state, those other iterations considered in parallel that presumed it would not be state changing are invalidated and expunged. The more processors that are available the more iterations that may be considered in parallel, thus the lower the chance of a 'step' (consisting of however many iterations may be considered simultaneously) occurring in which no state change takes place. Though the addition of extra processors yields diminishing returns, it moves the simulation closer to the optimum situation where every step results in a statechange thus maximising the parallelisation possible with this method. At this point further performance improvements would require a reduction in the number of MCMC statechanges required for the chain to converge - this is the domain of statisticians and the writers of the state-change proposer, thus is beyond the scope of this thesis.

So far we have considered each MCMC iteration to be of a fixed, constant realtime duration. In the following chapter we consider the consequences of variable realtime-duration iterations and present an extension to the speculative move

*Strictly speaking the prior and/or likelihood terms may individually contain calculations that can be conducted in parallel but this is not restricted to intra-move parallelisation, and can be just as easily applied to speculative moves.

method to accommodate such circumstances.

Chapter 4

Parallelisation by Speculative Chains

In the preceding chapter it was observed that whilst a Markov Chain must perform state changes in a strictly sequential order, those proposed state changes that are rejected have no impact on the final state of the simulation. A method called speculative moves was presented to compress a Markov Chain such that only those moves that were accepted consumed real time. This was achieved by considering a 'batch' of possible state changes simultaneously but allowing at most one of those potential changes to be used to effect an actual state change. The rejected iterations in each batch thus take negligible real-time to consider. Depending on the size of each batch of state change proposals the runtime required to perform a number of iterations may be reduced to (but not below) that required to perform only those iterations that would have resulted in a state change. Given suitable hardware this method can be effectively applied even when the time required to propose and consider a statechange is very small ($\sim 100\mu s$, or 10 000 moves a second). The effectiveness of this 'speculative moves' method is dependant on a high proportion of proposed changes being rejected, fortunately in practical MCMC applications a rejection rate of 75% is considered normal. Using the section 2.5 applications, reductions in runtime of 35 and 55 percent were obtained on SMP machines using two and four processors respectively.

In this chapter the speculative move concept is examined for applications where the time spent proposing and considering state changes varies considerably yet predictably. Naively applying speculative moves in such applications yields poor results, possibly even prolonging the simulation's beyond that of a simple sequential implementation. This is addressed first by refining the speculative moves implementation so less time is unnecessarily consumed, and then by replacing speculative moves with a speculative chain when one of the slow-processing state changes is proposed. Compare this with speculative moves, where should one of the state change proposals take a long time to process the entire batch would be delayed, with the end result remaining as at most one state change being applied. Using speculative chains, should the long-duration state change be rejected then the end-state of the speculative chain would be used as the new state of the primary chain.

4.1 Speculative Move Considerations

The preceding chapter assumes that a single value for mean processing time per move (τ) is adequate, and for the example simulations considered so far this is correct. However, there are applications where there may be substantial yet predictable variations in the time taken to process different types of move. Consider situations where the model being constructed contains composite structures such as trees (for example in the mapping of vascular trees as in [21, 57]). There may be moves that operate on individual features over small areas of the image (such as fine tuning a single node in the tree) and operations that modify large composite structures spread across large portions of the image. Even without composite structures there may be moves with effects that are highly localised (thus cheap to compute the change to the likelihood and prior terms should the move be applied) and others that modify variables with a non-localised effect forcing a (computationally expensive) complete recalculation of the prior and likelihood terms.

Instead of a single mean move time τ for all moves, let us consider a situation where we have a set $\mathbf{M_f}$ of moves that can be processed rapidly in time τ_f and a set $\mathbf{M_s}$ that requires τ_s time to process, where $\tau_f \ll \tau_s$. For example, moves of set $\mathbf{M_f}$ will cause small alterations whose effect on the prior and likelihood terms can easy be calculated, whilst the moves of $\mathbf{M_s}$ result in more dramatic changes that require extensive or complete recalculations of the prior and likelihood terms. When using the *speculative move* mechanism as described in the preceding chapter the presence of set $\mathbf{M_s}$ moves amongst $\mathbf{M_f}$ moves causes inefficient processor utilisation. Consider one MCMC step with n threads. If at least one thread considers an $\mathbf{M_s}$ move, any thread that considers a $\mathbf{M_f}$ move must wait idle for $\tau_s - \tau_f$ whilst the $\mathbf{M_s}$ move completes processing. If the probability of any single MCMC iteration being a $\mathbf{M_f}$ move is q_f then the probability of a speculative move step taking time τ_s is $1 - q_f^n$ thus each step will on average take

$$\tau_f q_f^n + \tau_s(1 - q_f^n) \tag{4.1}$$

Combining this with equation eq. (3.8), the expected number of steps required, gives us a new expression for the predicted runtime for N iterations.

$$T = N\left(\tau_f q_f^n + \tau_s(1 - q_f^n)\right)\frac{1 - p_r}{1 - p_r^n} \tag{4.2}$$

Figure 4.1 shows this plotted for varying q_f with common values of n (1,2,4,8), $p_r = 0.75$, and each long moves taking five times the processing time of a typical short move. The y-axis is the normalised runtime, such that '1' is the time taken for the sequential program to complete a fixed number of iterations with no $\mathbf{M_s}$ moves being proposed. The benefit of speculative moves (relative to equivalent sequential runtime) is of course identical if all moves performed are from the same set ($\mathbf{M_f}$ or $\mathbf{M_s}$, corresponding to $1 - q_f = 0$ and $1 - q_f = 1$ respectively). For values of q_f in between, there will be steps where both $\mathbf{M_f}$ and $\mathbf{M_s}$ moves are considered concurrently. In these steps, the thread(s) carrying out the $\mathbf{M_f}$ move(s) will be

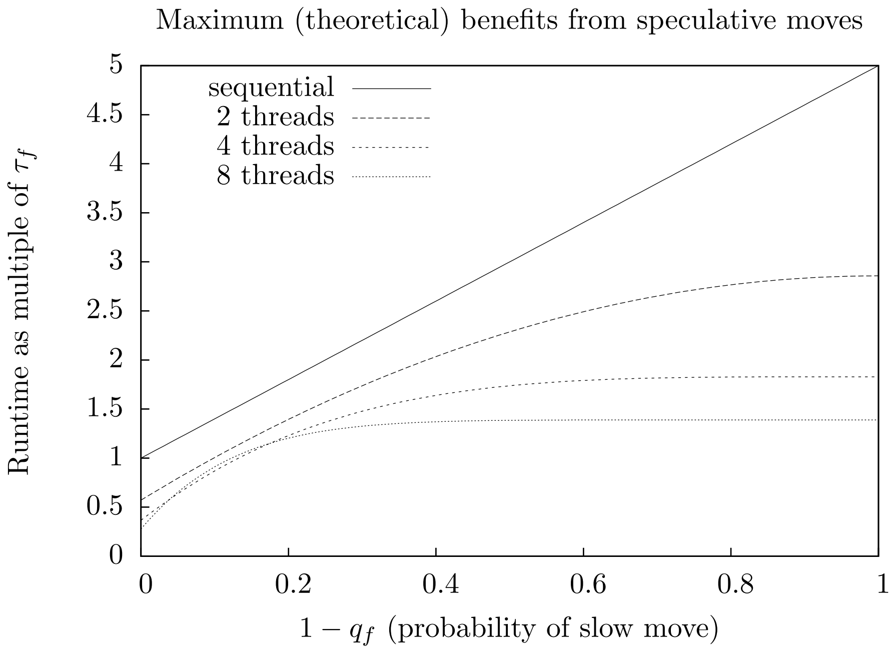


Figure 4.1: The impact of long running moves on speculative move runtime. $p_r = 0.75$, $\tau_s = 5\tau_f$

idle for time $\tau_s - \tau_f$ as they wait for the $\mathbf{M_s}$ move(s) to complete before continuing with the next set of speculative moves. The presence of this idle time means the runtime-reducing effect of speculative moves is impaired, although (in this case) the speculative move implementations do not become slower than the sequential version. The benefit provided by multithreading is reduced, instead of providing a runtime reduction of $\approx 43\%$ the two threaded version only reduces runtime by $\approx 22\%$ when 20% of moves are $\mathbf{M_s}$. As $\mathbf{M_s}$ moves become the norm ($q_f \to 0$) the full benefit of speculative moves is of course restored although always at a net increase in runtime.

In the more extreme case of fig. 4.2, where $\mathbf{M_s}$ moves take the time of 100 $\mathbf{M_f}$ moves, the benefits of speculative moves are lost though the presence of comparatively few $\mathbf{M_s}$ moves (note that the scale along the x-axis only goes up to a $\mathbf{M_s}$ move proposal probability of 0.02). Obviously the presence of $\mathbf{M_s}$ moves is going

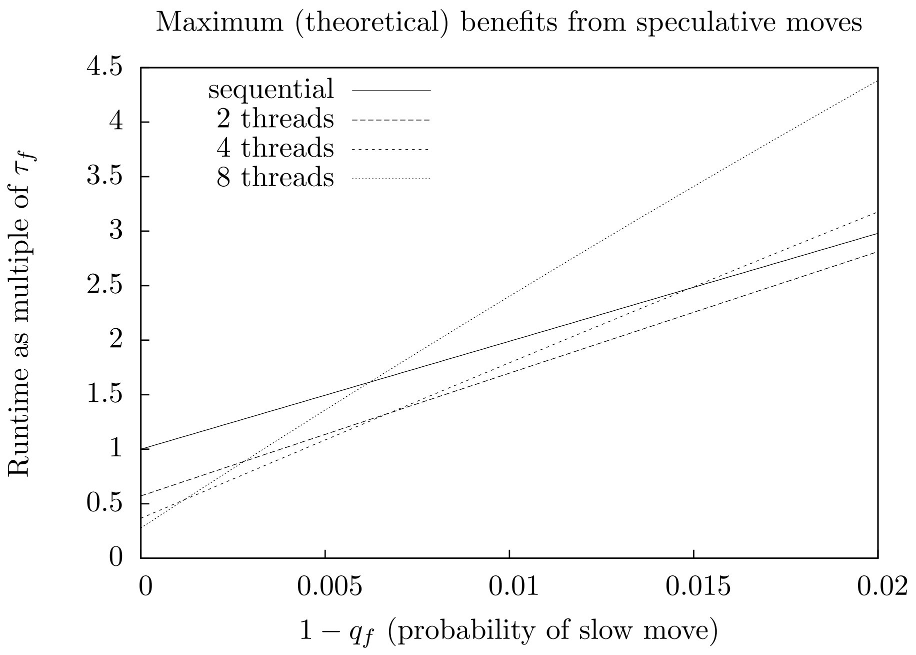


Figure 4.2: The impact of long running moves on speculative move runtime. $p_r = 0.75$, $\tau_s = 100\tau_f$.

to increase the runtime but when speculative moves are used the effect is disproportionately large for even small values of q_f. If just 1.5% of moves are of the long duration variety, all benefits of four-thread speculative moves are lost, increasingly $\mathbf{M_s}$ moves and speculative moves becomes a hindrance until at least 25% of moves are from $\mathbf{M_s}$.

4.2 Improving Speculative Moves

The presence of $\mathbf{M_s}$ moves has a detrimental effect when using speculative moves because they impair thread utilisation, as shown in fig. 4.3 a). In each program cycle involving a $\mathbf{M_s}$ move, threads performing a $\mathbf{M_f}$ move are left idle whilst they wait for the slow move to complete. The naive implementation of speculative moves

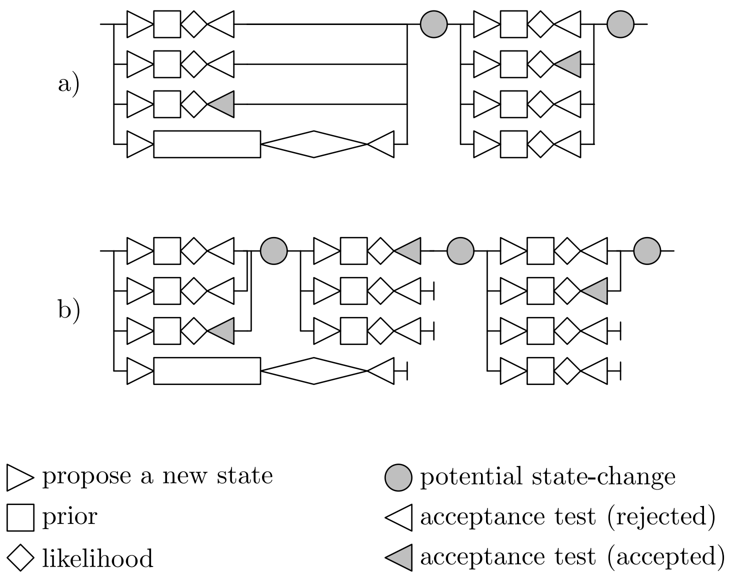


Figure 4.3: Each row represents a thread, vertical lines indicate synchronisation points between threads, shapes represent work being done. Time passes from left to right. a) The presence of long-running moves reduces the benefits of the 'naive' speculative move implementation. b) By using threads only if they are not already busy we mitigate the adverse effect of longer-than-normal moves.

presented earlier (fig. 3.2) guarantees that all speculative moves will be employed at each loop round the program cycle by synchronising the threads (waiting for all move calculations to complete) before starting the next set of move proposals. The threads are always used for each step, delaying the next step if just one thread is busy working (whether the results of that thread will be used or not). The alternative is to use the threads lazily, a thread will only be used for a step of the program cycle if that thread is available when it is needed.

Under this revised implementation if a proposed move is rejected we will wait for the speculative move(s) to make decisions and act accordingly (as before). However, when a proposed move is accepted any additional speculative move threads that are active are flagged as cancelled, then the primary thread immediately begins work on the follow-up move. When a new speculative move needs to be processed, any threads that are flagged as ‘cancelled’ but have not yet ceased processing are ignored and for that program cycle fewer speculative moves than normal are used. Speculative moves are considered only if there is a thread ready and waiting to be used, a speculative move will not be employed if it delays work on moves that are guaranteed to count towards the total number of MCMC iterations performed. Since the maximum number of speculative moves may not be utilised if one or more threads are busy, the average number of ‘normal’ MCMC iterations performed in each more steps (loops round the program cycle) is reduced. More steps will be required to obtain the same number of MCMC iterations, however the average time per step will be decreased as it will no longer be necessary to wait for invalidated $\mathbf{M_s}$ moves to complete their (unnecessary) processing. The net result is a increased number of normal MCMC iterations performed per unit time.

Figure fig. 4.3 b) shows lazy thread use in action. In the first step shown the fourth thread is taking longer than normal to complete, either the move being considered is from $\mathbf{M_s}$ or processing was delayed by resource conflicts (i.e. a background process was temporarily allocated control of a processor core). Since the move considered on the third thread has been accepted, there is no need to wait for the results of the fourth thread to complete, so it is flagged as cancelled. Once the move from the third thread has been applied the next batch of moves is considered on the three available threads. When the fourth thread finally does complete processing it discards its results (they are no longer relevant) and reverts to its idle state to await and participate in the next batch of moves to be considered.

It would be preferable for the fourth thread to simply cease processing immediately upon the determination that its results are irrelevant, that way all threads would be available for the next step in the program cycle. Unfortunately this is not always achievable. Killing and restarting a thread in order to stop work on a move is not an option as it does not allow the thread to release any resources it was using, potentially causing memory leaks and/or deadlocks (if the thread holds a mutex lock that it has not yet released at the time of the threads demise). 'Terminating' a cancelled move requires the 'cancelled' flag for that thread to be polled throughout each move's time-consuming calculations, skipping the remaining calculations if the cancellation flag is set. Since this flag is shared between threads, access to it must be synchronised (reads/writes controlled by a pthread mutex), adding overhead even if the move is never cancelled. Frequent polling allows for a faster response to the cancellation flag being set, at the expense of the increased overhead in repeatedly checking the mutex-protected flag, and increased complexity in the move calculations in order to enable this polling to take place. The choice of whether it is worth enabling the premature termination of cancelled moves needs to be made on a case-by-case basis, taking into account the added difficulty of implementation, the frequency with which moves will need to be terminated, and whether the move can be terminated fast enough to make the added overhead worthwhile.

To assist in this decision, consider the case where no moves can be terminated prematurely. It is possible for all threads performing speculative moves to become 'occupied' by cancelled $\mathbf{M_s}$ moves, leaving just the primary thread to perform work and resulting in near-sequential runtime. This will only become an issue if $\mathbf{M_s}$ moves are proposed faster than they can be cleared from the threads performing speculative moves. A thread that is performing a $\mathbf{M_s}$ move will take the same time as $\frac{\tau_s}{\tau_f}$ fast ($\mathbf{M_f}$) moves to complete the $\mathbf{M_s}$ move. For the remaining threads to be kept clear of another $\mathbf{M_s}$ move whilst the first is still being processed, the next $\mathbf{M_s}$ move should not be proposed for $\frac{\tau_s}{\tau_f}$ iterations (the time it takes a slow move to process divided by the time it takes a fast move to process.). In other words the probability of proposing a $\mathbf{M_s}$ on any of the n threads should be less than $\frac{\tau_f}{\tau_s}$, giving

$$\begin{aligned} (1-q_f)^n &< \frac{\tau_f}{\tau_s} \\ q_f &> \sqrt[n]{1-\frac{\tau_f}{\tau_s}} \end{aligned} \tag{4.3}$$

where q_f is the probability an arbitrary move belongs to $\mathbf{M_f}$ as oppose to $\mathbf{M_s}$. This means less than $< 10\%$ of moves can be slow (from $\mathbf{M_s}$) when a slow move is 5 times the length of the fast one, or only $< 0.5\%$ of moves if slow moves take the time of 100 fast moves. Implementers are therefore encouraged to accommodate the early cessation of processing $\mathbf{M_s}$ moves as they design the move proposal and prior/likelihood implementation.

4.3 Speculative Chains

In the preceding section we have reduced or eliminated the impact of invalid (cancelled) $\mathbf{M_s}$ moves on the program runtime by not forcing the whole program cycle to wait for threads that are temporarily unavailable/busy, and/or by causing the $\mathbf{M_s}$ moves to stop processing early if and when they are made irrelevant by the acceptance of another move. We will now address the bottleneck caused by *necessary* $\mathbf{M_s}$ moves by extending the speculative move philosophy. Let there be n threads labelled 1 to n in order of priority, thread 1 being the primary thread and threads 2 to n performing speculative moves (in descending order of preference). If thread i is considering a move from $\mathbf{M_s}$, all threads $> i$ that consider a $\mathbf{M_f}$ move will be idle for $\tau_s - \tau_f$ whilst they await a decision to be made on i's move, as shown by fig. 4.5 a). If the i'th thread's move is rejected this idle time is a waste. Should i's move be accepted it is irrelevant - their results will be discarded as the i'th thread will enact a statechange invalidating any other speculative moves considered in that step.

To avoid unnecessary idle time whenever a thread i performs a $\mathbf{M_s}$ move we perform a speculative *chain* on thread $i + 1$. Instead of using this thread to propose and test a single move, we create a temporary clone of the current chain state and use the speculative chain to perform multiple MCMC iterations on this copy. If there are additional threads available this speculative chain can itself make of speculative moves, using all threads $> i + 1$, and potentially threads $< i$ as well, once those threads become idle after processing and rejecting the $\mathbf{M_s}$'s predecessors (were one of the predecessors accepted the $\mathbf{M_s}$ would of course be cancelled). This

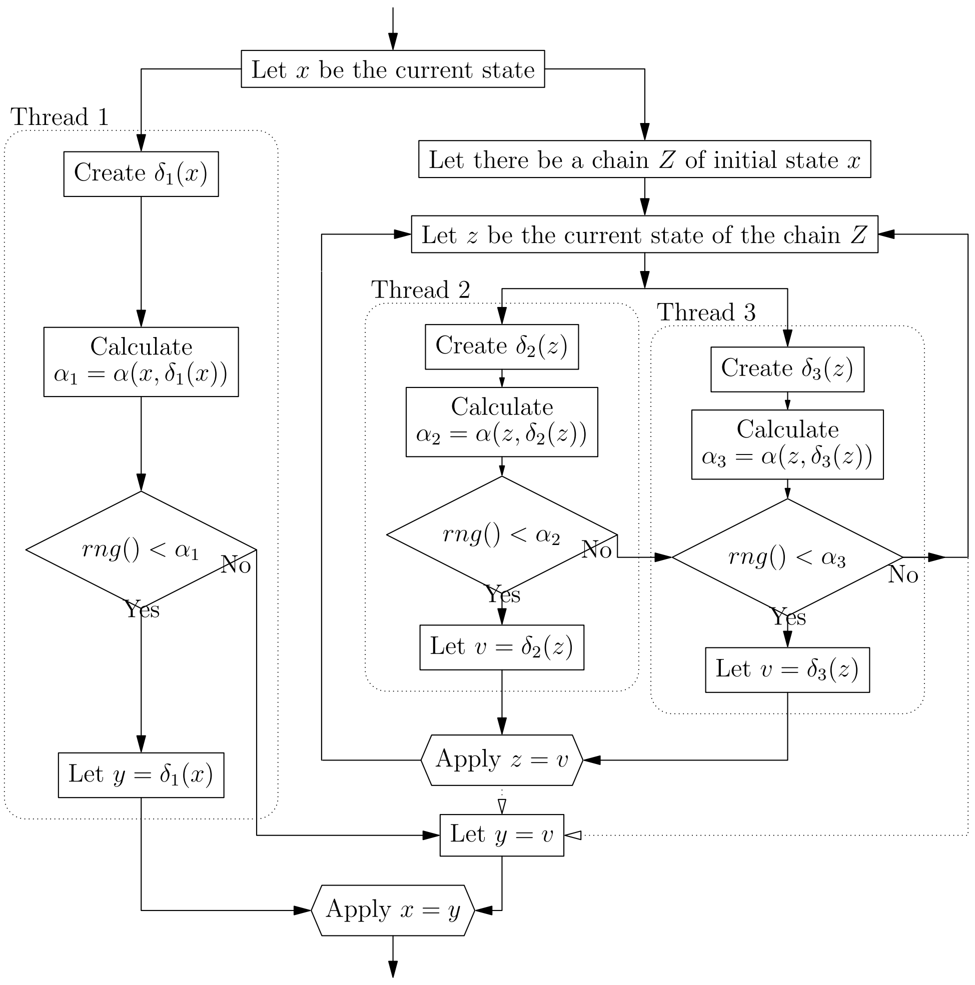

Let x be the current state
Thread 1
Create $\delta_1(x)$
Calculate $\alpha_1 = \alpha(x, \delta_1(x))$
$rng() < \alpha_1$
No
Yes
Let $y = \delta_1(x)$
Let there be a chain Z of initial state x
Let z be the current state of the chain Z
Thread 2
Create $\delta_2(z)$
Calculate $\alpha_2 = \alpha(z, \delta_2(z))$
$rng() < \alpha_2$
No
Yes
Let $v = \delta_2(z)$
Thread 3
Create $\delta_3(z)$
Calculate $\alpha_3 = \alpha(z, \delta_3(z))$
$rng() < \alpha_3$
No
Yes
Let $v = \delta_3(z)$
Apply $z = v$
Let $y = v$
Apply $x = y$

Figure 4.4: Example program cycle using a speculative chain. This is the program cycle that will occur when thread 1 performs a long duration ($\mathbf{M_s}$) move.

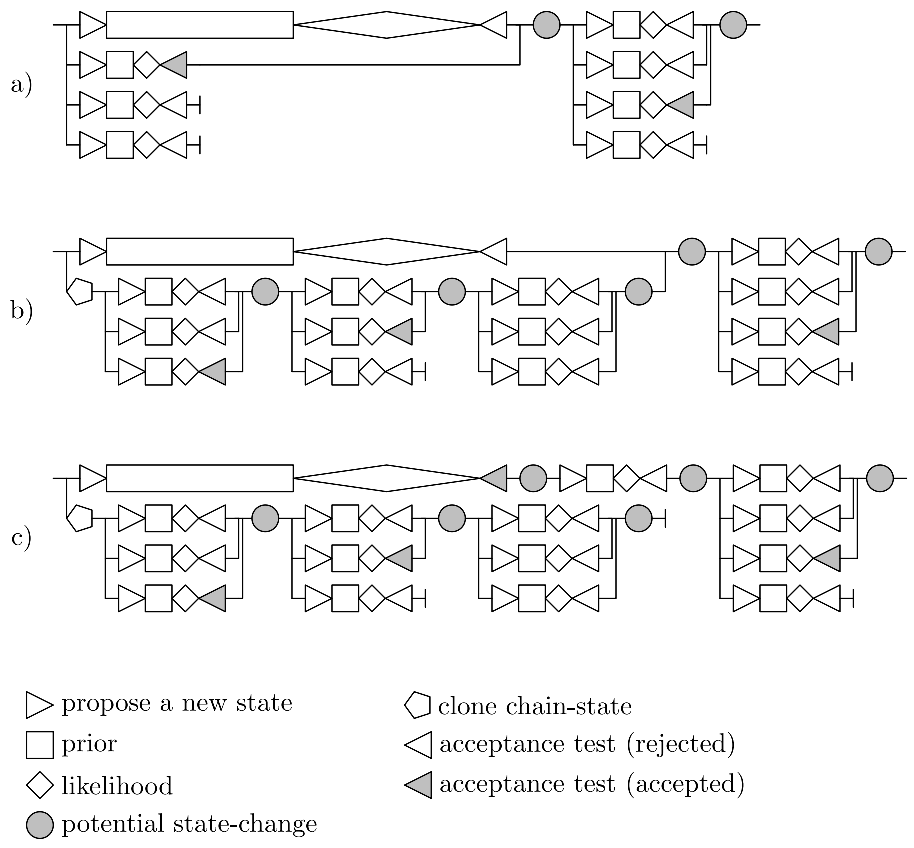


Figure 4.5: Each row represents a thread, vertical lines indicate synchronisation points between threads, shapes represent work being done. Time passes from left to right. a) Even with lazy thread use, the presence of a long-running ($\mathbf{M_s}$) move can still cause idle time on processors. b) A speculative chain allows useful work to be done on these processors by assuming the long-running move will be rejected. c) The speculative chain is discarded if the long-running move is accepted.

is illustrated in fig. 4.4, where a single program cycle is shown for the case when the first thread (1) conducts a $\mathbf{M_s}$ move. If thread i's move is rejected (as in fig. 4.5b) the state of the speculative chain will be used as the new state of the primary chain and we can return to the normal speculative move program cycle until the next $\mathbf{M_s}$ move is encountered. If thread i's move is accepted (as in fig. 4.5c) the speculative chain will be asynchronously messaged to terminate at the next opportunity whilst thread i continues MCMC processing as normal (whether the $> i$ threads are available for participation in the next round of speculative moves depends on the speed of their response to the termination signal).

To be more specific, consider the case where the speculative moves with speculative chains is applied on a system with two available processors. Let q_f be the probability with which a $\mathbf{M_f}$ is proposed, and p_r the probability that a move ($\mathbf{M_s}$ or $\mathbf{M_f}$) will be rejected. For each step in the program cycle there are four possible situations.

1. If the primary move is from $\mathbf{M_f}$ and accepted (occurring with probability $q_f(1-p_r)$) then the step takes only τ_f irrespective of whether a $\mathbf{M_s}$ or $\mathbf{M_f}$ move is considered speculatively (assuming the overhead of aborting a move is negligible). Either way, only one iteration is performed in that step.

2. If the primary move is from $\mathbf{M_f}$ and rejected (occurring with probability $q_f p_r$) then the step time is dependant on the type of move considered speculatively. On average the runtime will be $\tau_f q_f + \tau_s(1-q_f)$. Whatever type of move is chosen, two iterations will have been performed in that step.

3. A primary move from $\mathbf{M_s}$ that is accepted (occurring with probability $(1-q_f)(1-p_r)$) will naturally take τ_s time to perform the single MCMC iteration.

4. A rejected $\mathbf{M_s}$ primary move ($(1-q_f)p_r$) will still take τ_s time, but performs the number of iterations expressed in equation (4.4) +1.

The use of a speculative chain is only needed when a $\mathbf{M_s}$ move is being considered in an earlier thread in the same step, and even then only if the expected time that move will take (τ_s) is long enough to justify the potentially substantial overhead involved in cloning the state of the primary chain; there needs to be sufficient time for the chain-state to be cloned, and several speculative steps performed on this cloned-state before the original $\mathbf{M_s}$ move ends. The chain-state cloning overhead depends on how the cloning is implemented:

1. Immediate duplication. The primary chain state is duplicated in memory upon the creation of the speculative chain. Whilst expensive for large or complicated models, this is the simplest to implement.

2. Deferred duplication. The primary chain state is not immediately duplicated. Instead the speculative chain reads from the primary chain's state in the same way a speculative move would[*]. Once the speculative chain decides to accept a move is the primary chain's state duplicated. The speculative chain applies its first accepted move to this copy, and from then on works on the copy rather than the primary chain's state. This is cheap if it is likely the speculative chain will not find an acceptable move before its preceding thread completes processing of the $\mathbf{M_s}$ move, i.e. p_r is high and $\frac{\tau_s}{\tau_f}$ is small. The downside is the less predictable processing time and the added complexity: whilst the chain state cloning procedure is underway the speculative chain may receive an abort request, and/or the primary chain state may be changed due to the action of an accepted speculative move (applied to the primary chain). If it is highly probable the speculative chain will accept at least one move, option (1) will be preferable.

3. Virtual duplication. The primary chain state is not duplicated in memory,

[*]Note that the primary chain's state will not change whilst this is taking place as it is busy considering its $\mathbf{M_s}$ move, and once the move on the primary thread is rejected or accepted the speculative chain will be stopped.

instead a record is kept of each of moves the speculative chain accepts. The speculative chain's state is implemented as the primary chain's state viewed through a 'filter' that takes account of the sequence of accepted statechanges. For example, a request for information on a particular feature in the speculative chain's state would actually interrogate the equivalent feature in the primary chain's state, then check through the list of moves accepted by the speculative chain to see if that feature's data would have been altered. Instead of the large upfront processing cost of cloning the primary chain's state the overhead is applied to each iteration of the speculative chain, with the overhead increasing with each move that the speculative chain accepts. Whilst the hardest to implement, this is potentially the most efficient when dealing with very large models that are too expensive to copy completely, provided the chain of accepted moves does not grow so long that the per-iteration overhead becomes significant.

Whilst immediate duplication is preferred for its simplicity of implementation, deferred duplication will be a good idea if the probability of a chain enacting at least one proposed move is still small. For large states (i.e. phylogenies that may be megabytes in size [3]) virtual duplication may be the only viable option, though that depends on the relative expense of copying the state compared to that of considering the $\mathbf{M_s}$ move.

Whilst the overhead of cloning (or simulating the cloning) of the chain-state may be substantial and (as with speculative moves) runtime benefits will only be obtained if the primary move is rejected (if accepted, all the calculations performed speculatively are discarded), speculative chains can nonetheless yield a significant performance improvement. In the time it takes the original $\mathbf{M_s}$ move to complete there is the potential for performing up to

$$\frac{\tau_s}{\tau_f q_f + \tau_s(1 - q_f)} \tag{4.4}$$

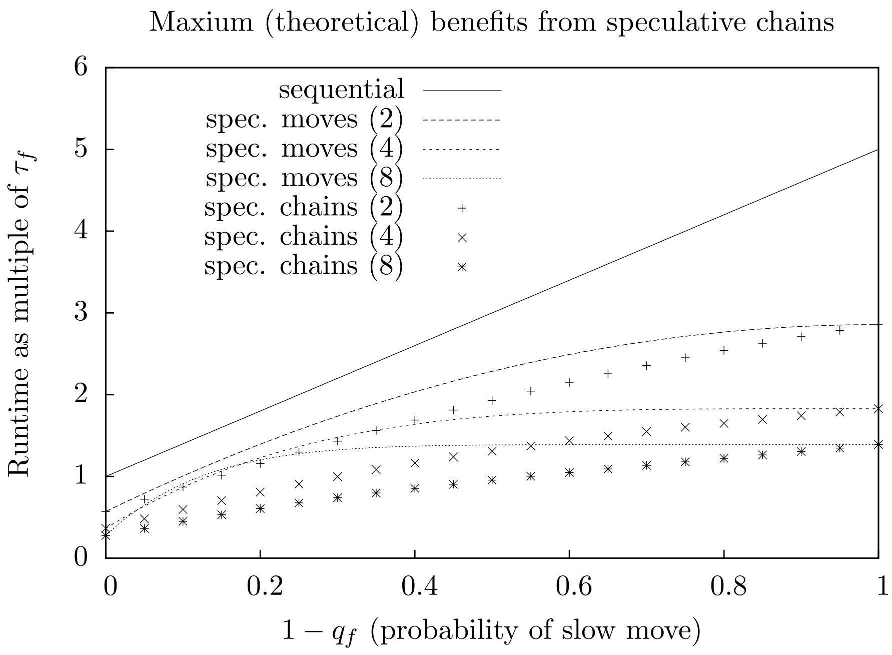


Figure 4.6: The impact of long running moves on speculative chain runtime. $p_r = 0.75$, $\tau_s = 5\tau_f$

sequential iterations within the speculative chain. Furthermore, if there are remaining unused processors the speculative chain may itself utilise speculative moves and chains, further boosting the number of iterations performed whilst the $\mathbf{M_s}$ move is considered. Should the original $\mathbf{M_s}$ move be rejected this entire chain of moves will be accepted. Contrast this with using speculative moves, only a single move would be considered whilst waiting for the $\mathbf{M_s}$ move to complete., should that solitary move be from $\mathbf{M_f}$ then the thread performing that move would be idle for the time $\tau_s - \tau_f$.

4.3.1 Theoretical Gains

Although we can use this information to derive a formula for the predicted runtime using speculative chains methodology, it is simpler (particularly when dealing with

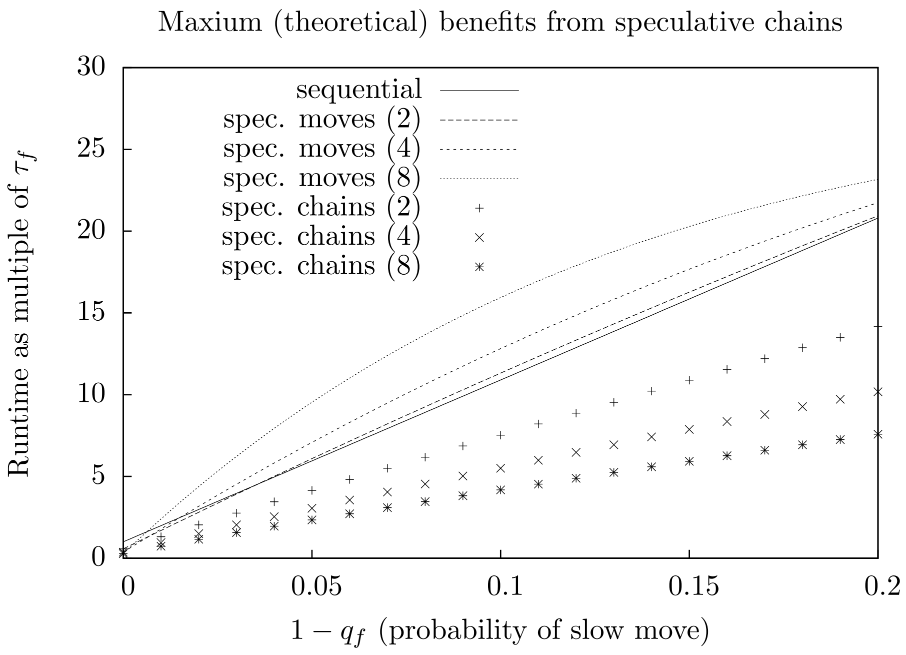


Figure 4.7: The impact of long running moves on speculative chain runtime. $p_r = 0.75$, $\tau_s = 100\tau_f$.

the 4, 8, or more threaded versions of the problem) to construct a simulator to loop through and sum up the expected runtime of each program cycle, accounting for the presence of speculative moves and threads. This simulator, implemented in Java and running on a Q6600 machine is capable of simulating $1.5x10^6$ 4-threaded program cycles a second. Some results from this simulator are shown in fig. 4.6 and fig. 4.7. The speculative move lines are the results obtained from eq. (4.2) whilst the speculative chain results are those from the simulator, the numbers in brackets is the number of processors/processing cores available. Along the x axis is the probability/frequency by which $\mathbf{M_s}$ moves are proposed. The y axis shows the normalised runtime, a value of '1' is the time the sequential code would take if there are no $\mathbf{M_s}$ (slow) moves proposed. The same assumptions used in the formulaic predictions of the previous chapter apply, the multithreading overhead is considered negligible so only move calculations are considered time consuming. Additionally we assume that

1. long-running moves may be aborted (cease processing) with negligible cost, when they have been invalidated by the acceptance of a move earlier in the order by which moves are considered[*]

2. a single value for the move rejection probability (p_r) holds for both $\mathbf{M_s}$ and $\mathbf{M_f}$ moves (in this example $p_r = 0.75$)

As shown by fig. 4.6 the use of speculative chains can provide a substantial performance over speculative moves if there exist moves that (predictably) take only 5 times longer than normal. If the difference between the $\mathbf{M_s}$ and $\mathbf{M_f}$ moves is greater (such as by x100 as in fig. 4.7 - note the different scale along the x-axis) it takes only a small percentage of moves to be in the $\mathbf{M_s}$ set for the speculative chains method to yield substantial results. In both cases the curves for simulated speculative chains are much flatter, retaining most of the benefit of speculative moves almost irrespective of proportion of $\mathbf{M_s}$ moves (though note that as with the speculative moves chapter, these predictions represent the upper bound on performance improvements).

4.4 Results

The circle-detecting program used for testing does not normally have sufficient variation in the processing time of different move types to test the speculative chain system. To obtain the time-per-move characteristics required for testing speculative chains, an alternative alter position move[†] was introduced that disallowed the likelihood optimisation from section 2.6, forcing the image likelihood to be recalculated

[*]Recall that speculative moves must be considered in a fixed, predetermined order irrespective of the order in which those moves complete processing.

[†]a move that changes the (x, y) coordinates of the centre of one of the circles, see section 2.5.1

from scratch thus lengthening the move consideration time (to around two or three times the normal time when dealing with images with 300 features)[*]. To simulate a larger disparity between move-processing times (such as a forty-fold difference between slow and fast moves) a loop was inserted to force the likelihood calculations to be repeated multiple times. In the field, such differences in the move consideration time (the difference between $\mathbf{M_s}$ and $\mathbf{M_f}$ moves) will be down to composite moves effecting large portions of the model (for instance, remove or modify an entire connected component of features as in 'delete tree' moves in [21]), the presence of more advanced logic in certain moves (for instance, a guided placement of new features rather than proposed new features being located entirely at random), or simply complete prior/likelihood recalculation where the calculation of move deltas for those values is prohibitively complex.

The benefits of speculative chains can be seen in fig. 4.8. This shows the runtime of the algorithm shown in section 2.5.1 working on autogenerated 1024x1024 images containing 300 circles, where $p_r = 0.75$ and $q_f = 0.999$ (0.1% of moves are from the $\mathbf{M_s}$ set). Despite less than 0.1% of moves being from $\mathbf{M_s}$ and τ_s being only 3-5 times τ_f, the speculative moves mechanism is rendered ineffective. Supplementing speculative moves with a single speculative chain whenever an $\mathbf{M_s}$ move is considered on the primary thread substantially improves performance across all architectures that were tested. When four threads are available, just using speculative moves yields the same results as speculative moves with two threads (not shown in fig. 4.8). Speculative chains were tested using four threads in two different ways. Firstly allowing at most one speculative chain to be active at any one time, secondly allowing as many speculative chains as will fit (in this case, three). In both cases, chain states were cloned using immediate duplication (copying the entire state for each new speculative chain). Results were mixed, whilst the Q6600 yielded results

[*]The presence of a move that performs a complete recalculation of the prior and likelihood terms is actually a sensible precaution to take against the slow drift of the perceived likelihood and prior from their 'real' value as a consequence of accumulated rounding errors.

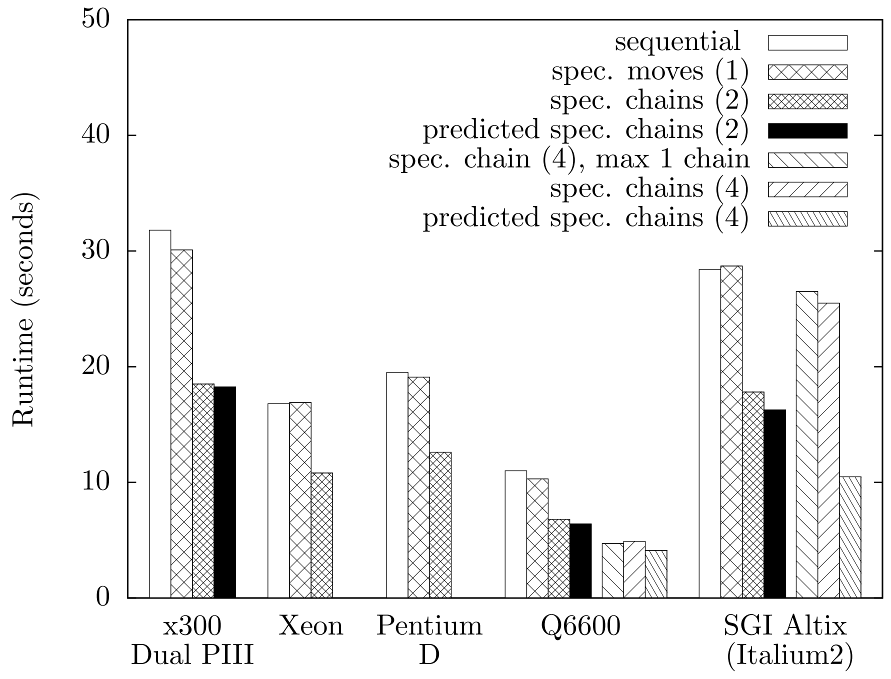


Figure 4.8: Comparison of speculative chains across architectures. $p_r = 0.75$, $q_f = 0.999$, $\tau_s = 5\tau_f$

close to those predicted, the SGI Altix seems ill-suited to this 4 threaded speculative chains parallelisation as the results produced were only marginally better than the sequential program and worse than when using 2 threads. This can be explained by the architecture of the Altix: 56 processors arranged in pairs, each pair with a shared memory cache. Memory accesses that cannot be resolved using cache are disproportionately expensive (in part as main memory must serve requests from all the processors). In this case on the Altix, the benefit of 4-thread speculative chains does not counter the overhead involved in cloning the chain's state.

The degree to which the different architectures achieved their potential runtime improvement is displayed in table 4.1. Unlike the results for speculative moves (table 3.3) these results are not grouped by their multithreading capabilities (dual-processor/dual-core/quad-core). This is because the overhead imposed by multi-

	2 threads	4 threads
x330 Dual-Processor	98	-
Q6600 (2x dual core)	91	88
Q6600$\star$ (2x dual core)	89	70
Altix (56 processor)	87	16

Table 4.1: The percentage of the potential (theoretical) reduction in runtime that was achieved for a set of experiments where moves are rejected with probability 0.75 and 0.1% of moves are of the 'slow' variety. Machines with higher values in the table are making more efficient use of their multiple processors.

	τ_f	τ_s	$\frac{\tau_s}{\tau_f}$
x330 Dual-Processor	1.59×10^{-3}	6.75×10^{-3}	4.23
Q6600 (2x dual core)	5.62×10^{-4}	2.53×10^{-3}	4.5
Q6600$\star$ (2x dual core)	4.29×10^{-4}	2.15×10^{-3}	5.02
Altix (56 processor)	1.42×10^{-3}	4.18×10^{-3}	2.94

Table 4.2: The difference in the ratios of the average time taken to perform fast and slow moves for a program for finding 300 circles in a 1024x1024 images, where moves are rejected with probability 0.75 and 0.1% of moves are of the 'slow' variety.

threading is small compared to the timeframe concerned (a single chain will last three to five times longer than an ordinary move in this example), the main factor is the additional workload involved in cloning the MCMC chain's state at the start of each news speculative chain.

The predicted values in fig. 4.8 were obtained using the simulator from section 4.3.1 and measurements from the time spent performing each of the different types of move. Those move-time measurements are listed in table 4.2. The instrumentation required to measure the move times was disabled during the timed program runs used to generate results such as fig. 4.8 (to avoid interference with the parallelisation mechanism) thus the average time per type of move (and therefore the full-program runtime predictions that are based upon this figure) may be slight overestimates. Also note that the simulator assumes that cloning the chain-state takes negligible time whilst the actual application uses immediate duplication.

Table 4.2 shows there is an added complication when comparing architectures on which to use speculative chains. Not only will the time per (sequential) MCMC iteration change depending on the hardware, but the ratio $\frac{\tau_s}{\tau_f}$ will as well. This difference in scaling between fast and slow moves is caused by the way the additional workload present in $\mathbf{M_s}$ moves is processed on different hardware and software environments, specifically differences in compiler optimisations, kernel efficiency[*], memory/cache latency, and build-in hardware optimisations (such as pipelining within the processor).

To illustrate some of the problems that can arise, tables 4.1 and 4.2 contain results for two different Q6600 machines. The data labelled Q6600 is from the machine mentioned in section 2.7, running Linux 2.6.18-36, and with the test program compiled using the GNU compiler GCC version 4.1.1. Q6600$\star$ is a separate machine running the Linux 2.6.27-11-server[†] and using GCC version 4.3.2. Unsurprisingly the machine with the newer compiler, kernel version, and server-optimised kernel performs the move computations faster, although from table 4.1 it implemented speculative moves less efficiently than its 'slower' equivalent, implying that there is a bottleneck whose rate of progress was less effected by the differing software configuration. A contributing factor to this bottleneck is the higher $\frac{\tau_s}{\tau_f}$ ratio of the Q6600$\star$.

Figure 4.9 shows the processing of the same data as fig. 4.8 but for varying $p_r^{\mathbf{M_s}}$ (the probability of any one move being a member of $\mathbf{M_s}$) on just the Q6600 ma-

[*]Upgrading from an early 2.4 kernel to a modern 2.6 kernel halved the execution time of one of the test programs used in section 3.5. It is suspected this is mainly down to the improvements to the efficiency with which the 2.6 kernel handles mutexes (the most basic means of synchronising threads and handling concurrency). A detailed analysis of the number of and cost of mutex operations across kernel versions and platforms is a subject for future work.

[†]Many Linux distributions ship two versions of the linux kernel, a 'normal' kernel suitable for desktop use and 'server' variety that prioritises computational efficiency over real-time responsiveness.

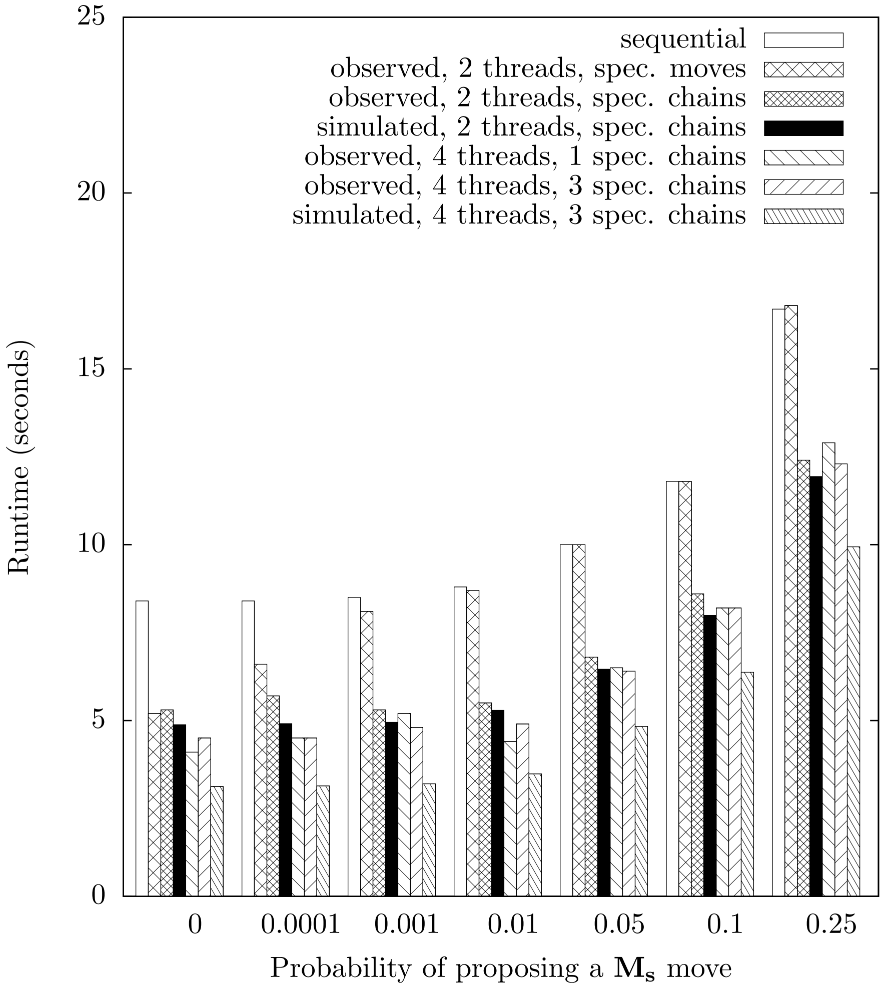


Figure 4.9: Altering $\mathbf{M_s}$ move reaction probability. $p_r = 0.75$, $\tau_s = 5\tau_f$.

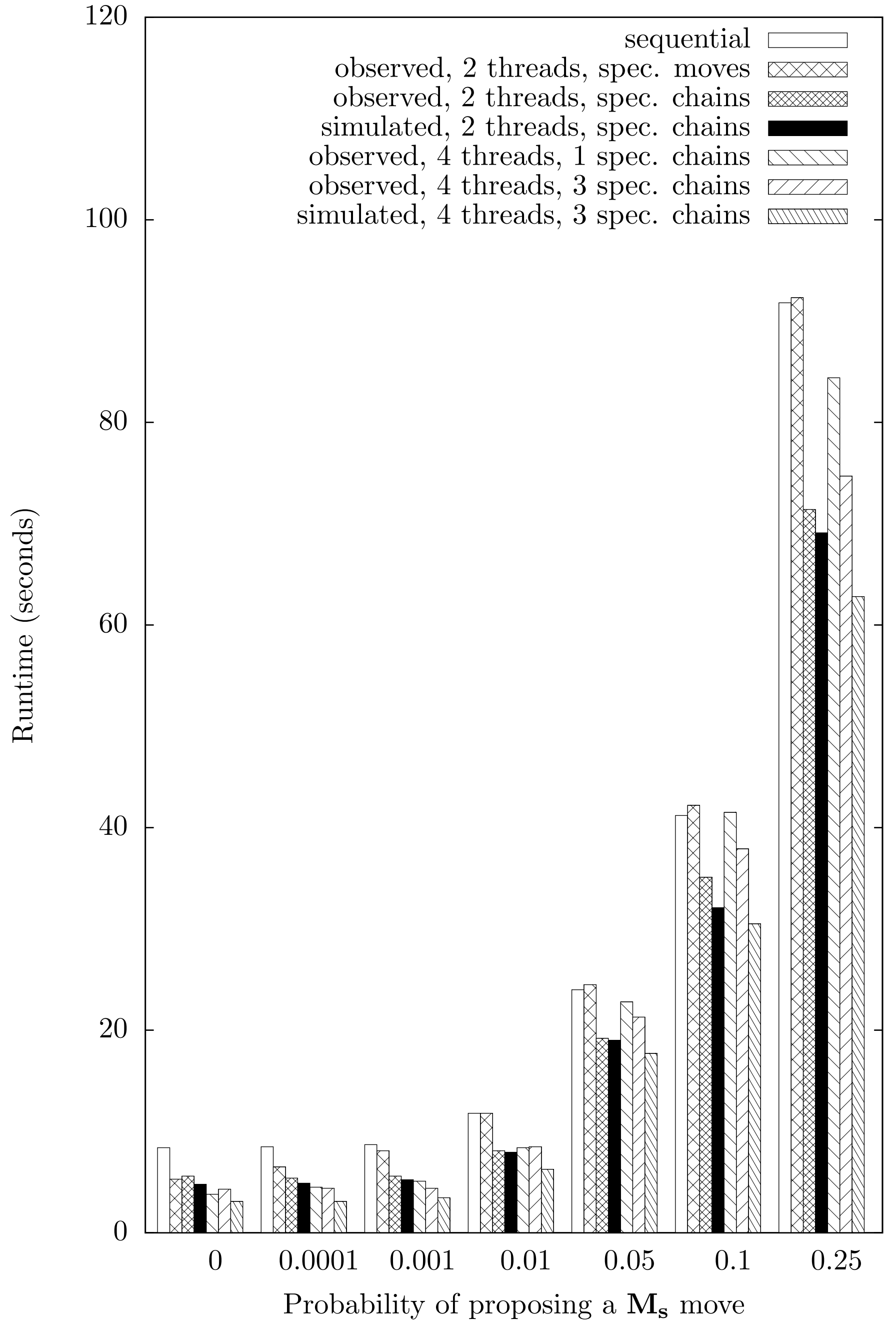


Figure 4.10: Altering $\mathbf{M_s}$ move reaction probability. $p_r = 0.75$, $\tau_s = 40\tau_f$.

chine, $\tau_s \approx 5\tau_f \approx 2ms$ (20 000 MCMC iterations total). If no $\mathbf{M_s}$ moves are present performance improvements from speculative moves are as would be expected from chapter 3. When $\mathbf{M_s}$ moves are proposed the benefit of speculative moves is lost (in this case when the percentage of $\mathbf{M_s}$ moves is somewhere in the range of 0.001% $\rightarrow$ 0.01%) and using speculative moves yields no performance improvement over the sequential program. Allowing a single speculative chain to be used (instead of a speculative move) when a $\mathbf{M_s}$ move is proposed allows the performance improvement from using speculative moves to be maintained despite the presence of $\mathbf{M_s}$ moves. Under the conditions used in this test the use of a single thread to perform speculative move/chains results in a reduction of runtime by around 33% from the sequential (or solely speculative-move-enabled) program, in line with predictions.

When four threads were available to the program using just using speculative moves yielded comparable results to using two threads with only speculative moves, the delay caused by the presence of $\mathbf{M_s}$ moves being limited to around that of the sequential implementation due to the ability to cancel such moves (the two thread and four thread speculative move runtimes exceed the sequential implementation due to overheads of multithreading and synchronisation, and because $\mathbf{M_s}$ cannot be cancelled 'immediately'). With the addition of speculative chains runtime was reduced to, at best, that of the 2-thread speculative move implementation regardless of how many speculative chains were permitted to operate simultaneously, falling short of predictions as $\mathbf{M_s}$ move become more frequent.

Figure 4.10 show the results for simulations using same parameters and hardware as for fig. 4.9 except $\tau_s = 40\tau_f$. For large values of q_f there is practically no difference. As q_f decreases (thus the proportion of moves from $\mathbf{M_s}$ increases) the slow executing moves force the program runtime to grow more rapidly. The proportion of the runtime reduced by the use of speculative moves compared to the sequential implementation is not substantially different from when $\tau_s = 5\tau_f$, though the larger sequential runtime for higher values of $1 - q_f$ make the performance improvement larger in absolute terms.

4.5 Summary

In this chapter the basic speculative moves method presented in chapter 3 has been extended to accommodate Markov Chain Monte Carlo iterations of variable (yet predictable) realtime duration. When there is significant variation in the processing duration between different types of iteration, there may be sufficient time to speculatively consider an entire chain of iterations whilst a single, long processing duration move is considered.

Whilst ignoring variations in processing duration between iteration types may result in a speculative move implementation that is substantially slower than the original sequential implementation, proper use of speculative chains can produce reductions in runtime exceeding the most optimistic predictions of plain speculative moves.

Speculative moves and speculative chains are entirely transparent to the statistical algorithm in use, although the implementation is different the end result is indistinguishable from a traditional sequential implementation[*]. To achieve more substantial reductions in runtime the MCMC algorithm must be altered, ideally in a way that has a minimal impact on the accuracy and rate of convergence of the simulation. The following chapter explores a number of means of doing so.

[*]Excepting the sequence of numbers obtained from any random number generators used by the algorithm

Chapter 5

Parallelisation by Partitioning

Chapter 4 examined the speculative move concept from chapter 3 in applications where the time spent proposing and considering state changes varies considerably yet predictably. Naively applying speculative moves in such applications yields poor results, possibly even prolonging the simulation's beyond that of a simple sequential implementation. This was addressed first by refining the speculative moves implementation so less time is unnecessarily consumed, and then by replacing speculative moves with a speculative chain when one of the slow-processing state changes is proposed. Compare this with speculative moves, where should one of the state change proposals take a long time to process the entire batch would be delayed, with the end result remaining as at most one state change being applied. Using speculative chains, should the long-duration state change be rejected then the end-state of the speculative chain would be used as the new state of the primary chain.

Having taken the concept of speculative execution of MCMC iterations as far it can go, we now consider how else a MCMC application may be parallelised. Implementing speculative moves and/or speculative chains makes no logical change to the MCMC algorithm, so these methods can be applied to any MCMC simulation without fear of any disruption in the results. To achieve any additional performance improvements more aggressive tactics must be attempted that may potentially alter the end results. This chapter explores a number of such methods, all of which are based upon the basic idea of processing different parts of the input data separately. As such, these methods are restricted to applications where some state changes have only a local effect, such as feature identification in images.

The first technique proposed is termed 'periodic parallelisation', and involves making the distinction between local state changes whose impact is limited to a small area of the image, and global state changes that must be considered as acting on the entire image. As its name implies, periodic parallelisation alternates between two modes, a global phase where only global state changes take place, and a local phase where only local state changes occur. For the local phase the image is split into multiple subimages using a randomly positioned grid, with local moves permitted to occur in different subimages simultaneously. By frequently switching between local and global phases and repositioning the local phase partitioning grid at each swap the long term impact of the partitioning is limited, ideally to the extent that it is negligible. This method is suited to many different parallel processing architectures, as the period between phase changes can be set so as to render the inter-process communication overhead insignificant (allowing this method to be used on SMP machines or over a cluster), although to prevent anomalies in the results the phase changes should occur as frequently as is feasible.

Although the exact consequence that periodic parallelisation has on the results is difficult to determine or predict, the concept is statistically sound. Somewhat harder to justify is the decision to split the original image and process the subimages as entirely separate entities until the very end of MCMC processing, at which point the results for the subimages are patched together. This is the concept of 'image splitting', covered in the latter half of this chapter. Two flavours of this method are presented, the first is termed 'intelligent', as it uses a pre-processor to divide the image such that no features of interest intersect any of the subimage dividing lines.

Obviously this relies upon such a pre-processor existing and being cheap to execute, but has the advantage that combining the results from each of the subimages is trivial. The alternative is blind image splitting, whereby the subimage boundaries are set irrespective of the image content and some heuristics must be employed to reconcile features occurring at the subimage boundaries (to facilitate this it is recommended that the subimages overlap to allow any controversial features spanning subimage boundaries to be clearly identified in both images). Both intelligent and blind image splitting require a non-MCMC algorithm to be developed (either for the pre-MCMC segmentation of the image or post-MCMC combination of subimages), and both suffer from the problem of allocating suitable prior variables to the subimages, although this is less of an issue if such prior variables are obtained from a pre-processor analysing the image rather than from the expectation that all images will have roughly the same properties thus using a single set of prior values for all images to be processed.

5.1 Parallelisation by Periodic Partitioning

The speculative execution parallelisation methods presented in the preceding chapters have been proven to make no change to the fundamental MCMC algorithm, only the manner in which it is implemented. Further parallelisation requires a more aggressive approach, but also requires we narrow our focus to input datasets that can be meaningfully partitioned, such as images. Since runtime increases significantly with the complexity and size of the image (more on this in section 5.2) the obvious parallelisation method is to break a large image up into partitions and consider each separately. Unfortunately this will cause artifacts along the partition boundaries as image features are not detected, imperfectly detected, or duplicated (detected in both partitions). Furthermore, it is not always the case that the *prior* assumptions concerning the full image hold when applied to subset of that image. Even though taken across a set of images, features may still be distributed at random; if we examine only a subset of one particular image it may well be the case that not only does the distribution no longer seem random, but the density of features (the number per unit area) may be substantially different to that of the entire image.

Despite these problems, in many cases it is possible to make use of this parallelisation-by-partitioning without impairing the statistical properties of MCMC. The basic idea is as follows. First, a number of MCMC moves that cannot be performed in parallel with any others are performed sequentially. The image is then randomly partitioned and a number of MCMC moves that can be performed in parallel are performed in each partition simultaneously, whilst ensuring that changes that could potentially affect the consideration of features in other partitions are forbidden. The changes to each partition are then combined back into a single model and the cycle repeats, with a number of the non-parallelisable moves being performed on whole image. The non-parallelisable 'global' moves will be making large-scale alterations to the image, whilst the parallelisable 'local' moves will be performing localised 'fine-tuning' of specific features. This cycle is repeated with sufficient frequency that the grouping of moves into a 'global move' phase and a 'local move phase', and the partitioning that takes place in the local phase, are statistically insignificant.

First we separate the moves that may be applied to the MCMC chain into two groups, global ($\mathbf{M_g}$) and local ($\mathbf{M_l}$).

$\mathbf{M_g}$ contains all moves that alter the configuration in a manner that impacts prior/likelihood calculations across the entire image/configuration. As such, a $\mathbf{M_g}$ move cannot be performed in parallel with any other move.

$\mathbf{M_l}$ moves make limited changes (akin to fine-tuning) whose impact is restricted to a small area and makes no changes to 'global' properties (such as the number of features in the configuration). Since the decision to accept or reject such a move is based solely on the image data in close proximity to the changed feature, multiple $\mathbf{M_l}$ may be performed simultaneously without violating the

MCMC criteria so long as the features modified by these moves are sufficiently distant. Such moves cannot overlap with $\mathbf{M_g}$ moves.

Under normal circumstances $\mathbf{M_l}$ and $\mathbf{M_g}$ moves are interleaved, preventing the parallel processing potential of $\mathbf{M_l}$ moves from being utilised. If we arrange matters so that batches of the potentially parallelisable $\mathbf{M_l}$ moves are proposed one after the other, then parallel processing can take place within each batch that batch of moves. To achieve this, at the start of each iteration instead of selecting a new proposed move at random from $\mathbf{M_g} \cup \mathbf{M_l}$ we alternate between performing z_g consecutive moves from $\mathbf{M_g}$ then z_l consecutive moves from $\mathbf{M_l}$, z_l begin chosen so that we preserve the long-term move proposal probabilities. If q_g is the probability of an arbitrary move being of a member of $\mathbf{M_g}$, then for some fixed number of iterations N

$$z_g = q_g N \tag{5.1}$$

$$z_l = (1 - q_g)N \tag{5.2}$$

To keep the probabilities of proposing moves from either of these two sets constant, if z_g moves are performed in each $\mathbf{M_g}$ phase then each $\mathbf{M_l}$ phase must perform

$$z_l = \frac{z_g}{q_g} 1 - q_g \tag{5.3}$$

moves. So long as the number of moves performed in each phase (of either $\mathbf{M_g}$ or $\mathbf{M_l}$ moves) is small compared to the total number of moves performed the fact that this alternating is taking place will not substantially alter the development of Markov Chain's state.

Now we are alternating between performing a batch of $\mathbf{M_g}$ moves and a batch of $\mathbf{M_l}$, the next step is to allow parallel processing within each batch of $\mathbf{M_l}$ moves. By our definition, consecutive $\mathbf{M_l}$ moves may be performed in parallel provided the moves are sufficiently distant from each other so as not to interfere with each others prior and likelihood terms. To achieve this we partition the image

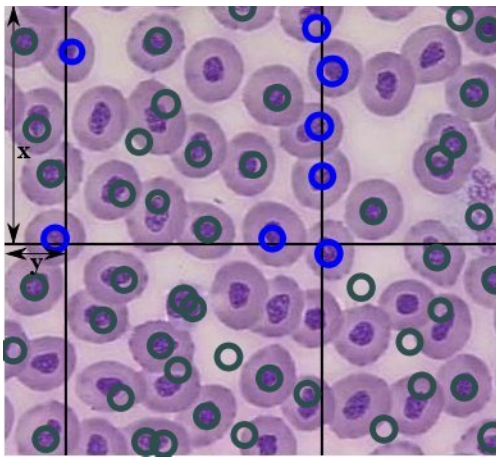


Figure 5.1: By frequently changing the offset of this partitioning grid we can prevent boundary anomalies caused by the partitioning persisting in the MCMC chain. The highlighted features intersect the partition boundary, so will be held immobile until a new offset is set for the partitioning grid.

with a uniform grid of spacing $x_{\max}$ along the x-axis and $y_{\max}$ along the y-axis. To avoid any potential conflicts between partitions, features whose prior/likelihood calculations take into consideration an area that intersects with the partition grid must not be selected for modification, and no feature may be created or moved such that any part of it (or its prior/likelihood considered area) intersects with or is outside its assigned partition. For example, with the circle-finding case studies from section 2.5 the likelihood of a circle is calculated from pixels on the circle's circumference and those contained within it, whilst the prior term for a circle only interacts with neighbouring circles if they overlap with it. Changes to the position or radius of a circle therefore only alter the prior/likelihood terms of other circles with which the altered circle intersects. Alter position or radius moves are therefore in $\mathbf{M_l}$, and may be performed on any circle that does not intersect the partition boundaries. Note that none of the moves that change the number of circles in the model (birth, death, merge or split) can take place in the partitioned phase as the total number of features in a model (λ) is a global variable found in the prior (see section 2.5).

$\mathbf{M_l}$ moves may now be performed in each area partitioned by the grid simultaneously. To avoid the partition grid imposing a long-term bias on the results (since features impacting prior/likelihood calculations in an area intersecting the grid are rendered unchangeable), for each batch of $\mathbf{M_l}$ moves performed a new x and y offset for the grid is chosen at random from the ranges $0..x_{\max}$ and $0..y_{\max}$ respectively, see fig. 5.1. With the offset of the partition grid being randomly reassigned for each $\mathbf{M_l}$ phase, over the long term the features will have an equal opportunity for modification by $\mathbf{M_l}$ moves. Assuming that the switch between $\mathbf{M_g}$ and $\mathbf{M_l}$ moves (and the accompanying redrawing of the partitioning boundaries) occurs sufficiently frequently there will be no persistent partition boundary anomalies, as there will be no persistent partition boundaries.

It has already been remarked that the number of iterations performed in the global and local phases must be set such that the overall move proposal probabilities are unaffected, the relationship between the number of moves in each local phase and the number of moves in each global phase being expressed in eq. (5.3). Additionally we need to split the number of iterations to perform during the $\mathbf{M_l}$ phase between each of the partitions. If all dimensionality-modifying moves[*] are in the set $\mathbf{M_g}$, each partition can be allocated the number of local iterations to perform in the same proportion as the number of model features contained within the partition's boundaries and that may be legitimately modified (not too close or intersecting with the partition boundary) compared to the number of such (modifiable) features taken across all partitions. If any dimensionality changing moves are in $\mathbf{M_l}$ it may be worth moving them to $\mathbf{M_g}$ anyway, otherwise certain partitions may perform more than their 'fair share' of iterations if features are not added/removed from

[*]Dimensionality-modifying moves are those that change the number of dimensions of the states-pace, i.e. by adding or removing a feature from the model.

all partitions at an equal rate (this depends on the actual distribution of features in the image, and the relative sizes of the partitions, as partitions along the image boundary will not be of full size).

5.1.1 Predictions

Given that the mean time to perform $\mathbf{M_g}$ (global) and $\mathbf{M_l}$ (local) moves are τ_g and τ_l respectively, how long will it take to perform N MCMC iterations using periodic parallelisation with s processors? With the probability that an arbitrary move will be in the set $\mathbf{M_g}$ set as q_g, the total number of $\mathbf{M_g}$ moves performed over the entire program run will be Nq_g. The total time spent in the global move phase is therefore

$$Nq_g\tau_g \tag{5.4}$$

Similarly $N(1-q_g)$ moves from $\mathbf{M_l}$ will be performed, however the processing of these will be spread over the s available processors. Assuming that the workload is evenly split between all the partition and that the parallelisation overhead is negligible, the actual time taken spent processing all the $\mathbf{M_l}$ phases will be

$$\frac{N(1-q_g)\tau_l}{s} \tag{5.5}$$

Giving the total runtime as

$$Nq_g\tau_g + \frac{N(1-q_g)\tau_l}{s} \tag{5.6}$$

which has been plotted in normalised form as fig. 5.2. The $\mathbf{M_g}$ move phase, not being partition parallelisable, is now the slowest component of the MCMC program. Although by definition the $\mathbf{M_g}$ phase contains moves that cannot be performed in parallel by partitioning, as a MCMC chain the $\mathbf{M_g}$ phase is susceptible to the application of speculative moves. The runtime of a periodic parallelisation program that uses speculative moves to accelerate the global move phases can be predicted by taking eq. (5.6) and replacing the expected number of $\mathbf{M_g}$ iterations (Nq_g) with the

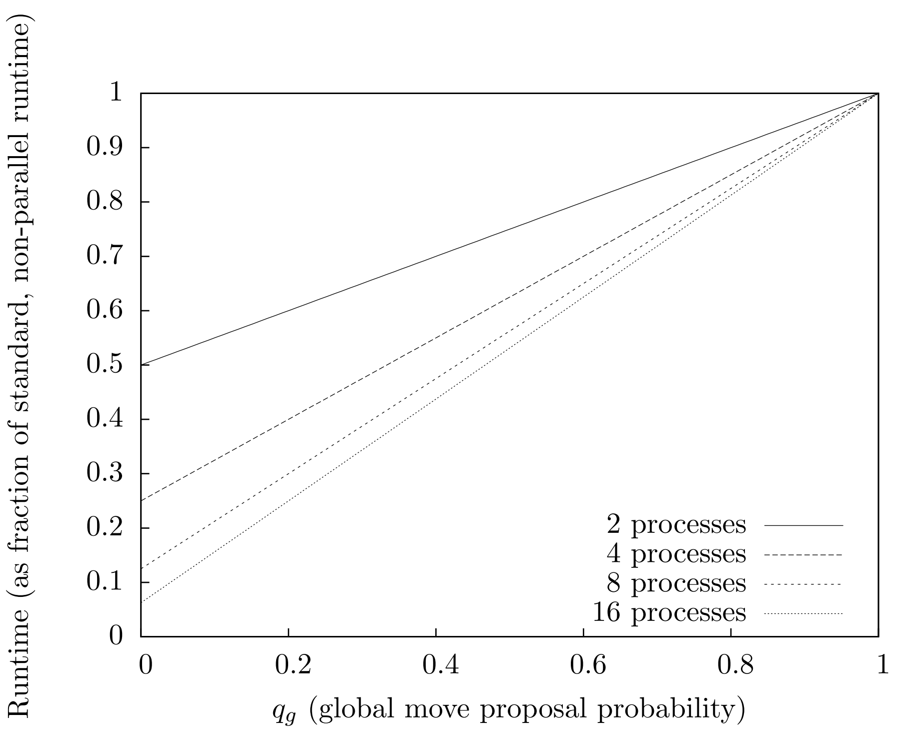


Figure 5.2: Predicted results for periodic parallelisation. $\tau_g = \tau_l$

expected number of speculative moves steps (eq. (3.8)), when using all the available processors (s) to perform speculative moves.

$$Nq_g\tau_g\frac{1-p_{gr}}{1-p_{gr}^s} + \frac{N(1-q_g)\tau_l}{s} \tag{5.7}$$

where p_{gr} is the probability that a $\mathbf{M_g}$ move will be rejected. Plotting eq. (5.7) in fig. 5.3 shows the predicted consequences of using periodic parallelisation during $\mathbf{M_l}$ phases and speculative moves during the $\mathbf{M_g}$ phases.

We can also use speculative moves to further increase the number of $\mathbf{M_l}$ moves that may be performed per unit time. Though it appears preferable to utilise any spare threads/processors to allow a greater number of partitions to be made and processed simultaneously, there is a limit as to how small a partition can be before no useful work can be done inside it. Since we prohibit any changes that may cause a feature nominally inside a partition to effect the prior or likelihood calculations

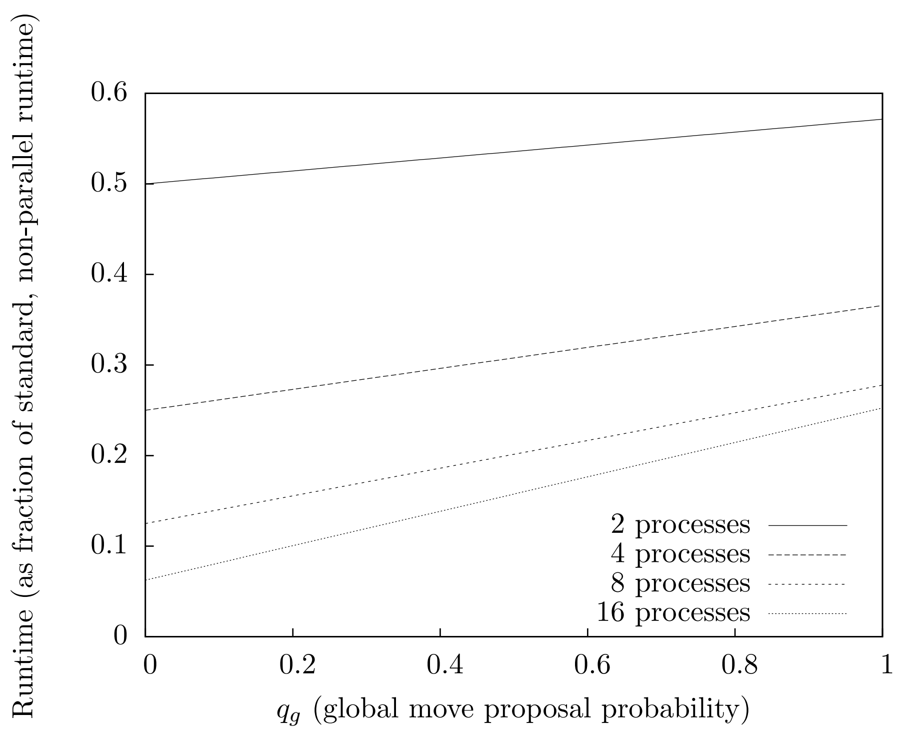


Figure 5.3: Predicted results for periodic parallelisation supplemented with global phase speculative moves. $\tau_g = \tau_l$, $p_{gr} = 0.75$

for any feature outside that partition, the area in which features may be changed is somewhat smaller than the area of the partition. For instance, consider the circle-finding algorithms of section 2.5 with the restriction (for the sake of simplicity) that there is no variation in circle radius, all circles have radius r. For $\mathbf{M_l}$ moves, the change in likelihood is determined only from those pixels touching or enclosed within the circle, and the prior term is changed only if another other circle intersects the changed one. Under these conditions, in a $\mathbf{M_l}$ phase only circles completely enclosed within the partition are subject to modification. Thus in a square partition of area x^2, circles can only be changed or relocated within an area of $(x - 2r)^2$ during $\mathbf{M_l}$ phases. To maximise the potentially modifiable area in each $\mathbf{M_l}$ phase (and reduce the potential disruption caused by using $\mathbf{M_l}$ and $\mathbf{M_g}$ phases) each partition should be substantially larger than the features to be found within it, i.e. $x >> 2r$. This can conflict with the desire to divide the image into as many partitions as possible in order to maximise the parallelisation performed. Determining the point where one concern dominates the other is a subject for future, application specific, research.

In the presence of spare processors, the use of speculative moves within each partition is a good alternative to increasing the number of partitions if we do not wish to shrink the partition sizes any further (again, the circumstances where one method is preferred over the other is a candidate for further application specific research). The use of speculative moves during the $\mathbf{M_l}$ phase may also be encouraged by system architecture. We note that if $\mathbf{M_l}$ phases are set to be long enough it becomes feasible to deploy each partition to a separate physical node (machine) in a cluster. If this is the case, and each node in the cluster also has true multithreading capabilities, it is natural to use separate physical nodes for each partition, and multithreading in each node for speculative moves. Starting with eq. (5.7) we can again apply eq. (3.8) (as we did to produce eq. (5.7)) to model the use of speculative moves on the chain of $\mathbf{M_l}$ moves performed within each partition. For a cluster with s nodes each with t threads, the best possible runtime is

$$N q_g \tau_g \frac{1-p_{gr}}{1-p_{gr}^t} + \frac{N(1-q_g)\tau_l(1-p_{lr})}{s(1-p_{lr}^t)} \tag{5.8}$$

Which gives predicted runtimes as shown in fig. 5.4, assuming that inter-node communication time has been rendered negligible by the number of iterations in each phase and time per each iteration. Note that we assume that partitions are still substantially larger than the features being detected.

In practice the frequency with which we alternate between $\mathbf{M_g}$ and $\mathbf{M_l}$ phases will also have an impact on the total runtime (recall the above predictions assume negligible overhead). Statistically we want these phases to be as short as possible to minimise any potential impact the partitioning may have to the short-term results. Practically we want each phase to be long enough to overshadow the overhead required in partitioning, distributing the workload to the parallel threads/machines, and the subsequent recombining the models. A similar balance must be made be-

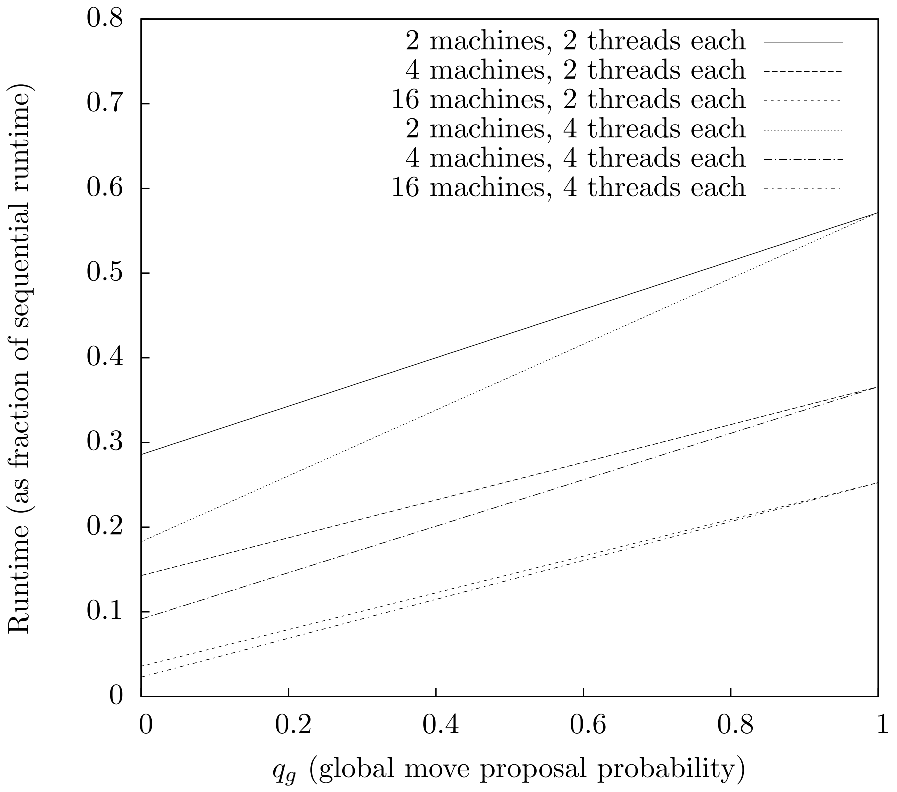


Figure 5.4: Predicted results for periodic parallelisation over a cluster, supplemented with global and local phase speculative moves. Threads are used for speculative moves, whilst nodes are used for periodic parallelisation. $\tau_g = \tau_l$, $p_{gr} = p_{lr} = 0.75$.

tween the number of partitions (more=faster) and the corresponding size of the partitions. More partitions mean each partition is smaller, which means the number of features that may be modified (and how those features may be modified) is more limited, thus more likely to delay the convergence of the MCMC algorithm.

Since different partitions will be allocated different numbers of iterations to perform (depending on the number of model features fully enclosed within each partition), the time taken to complete the assigned iterations will vary considerably (even if features are uniformly distributed, partitions along the edge of the image will inevitably be less than their full size, contain fewer than normal features thus be allocated fewer iterations to perform per local move phase). The processor deadtime that results can be reclaimed through the use of a task scheduler, allowing more partitions than there are available processors to be employed.

Finally, note that we are not limited to the speculative moves method, speculative chains may also be used to improve the performance of the $\mathbf{M_g}$ and $\mathbf{M_l}$ phases, if the selection of available moves within each phases warrants it (different types of move within a phase have substantially differing predicting runtimes).

5.1.2 Example

Let us first consider processing a 1024x1024 image containing 150 circles of mean radius 10. The global moves ($\mathbf{M_g}$) are those that add, delete, merge, split, or replace a circle[*]. The local ($\mathbf{M_l}$) moves are those that alter either the position or radius of a circle. The proposal probabilities are such that 60% of moves are from $\mathbf{M_l}$. The image will be split into four rectangular partitions using a single coordinate where all partitions meet. Whilst suboptimal in terms of processor utilisation when 4 processors are available (partitions will rarely be of equal size) this does minimise overhead from splitting and merging configurations by keeping such operations simple.

[*]The 'replace' move relocates a circle to a random position in the image, its purpose here is to provide long distance moves across the image.

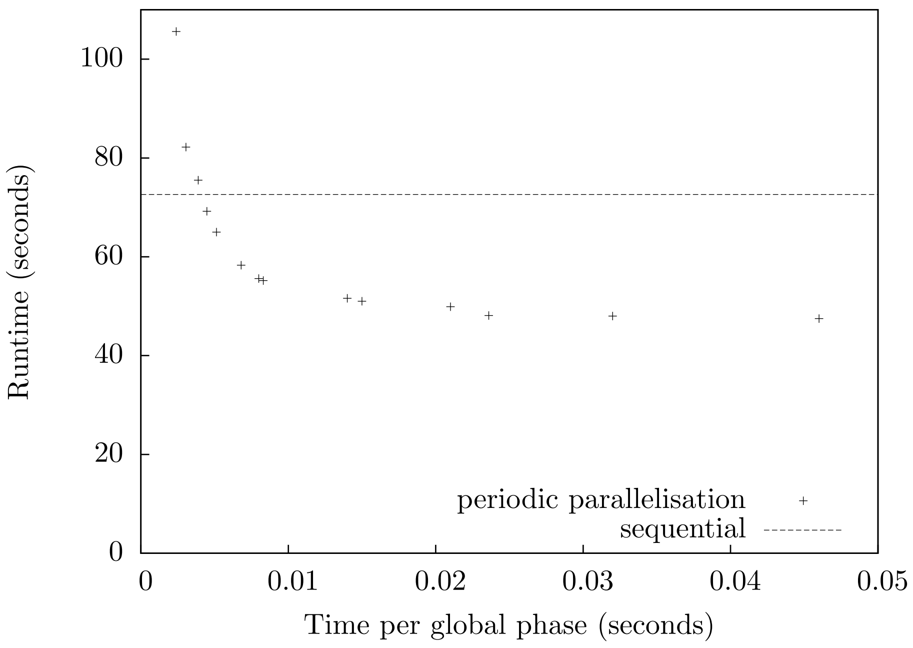


Figure 5.5: Example of periodic parallelisation on 1024x1024 images with only four partitions, run on a Q6600. The horizontal line represents the runtime of the sequential implementation.

Using these parameters, fig. 5.5 shows the time taken to perform a fixed number (500 000) of MCMC iterations for different frequencies of repartitioning, the horizontal line representing the runtime of the sequential implementation. In this case each global move phase must last at least 4ms ($\sim$ 23 iterations) for the periodic parallelisation method to be faster than the sequential implementation, any less than this and each local move phase does not last long enough for the benefits of parallelising $\mathbf{M_l}$ moves to outweigh the costs. Once each global phase takes 20ms or more there is no substantial runtime improvement to longer phases, it is at this point that the costs of parallel processing of $\mathbf{M_l}$ moves cease to be a significant proportion of the runtime. This equates to around 130 iterations, and thus (using eq. (5.3)) each local phase will perform 194 iterations spread amongst all the parti-

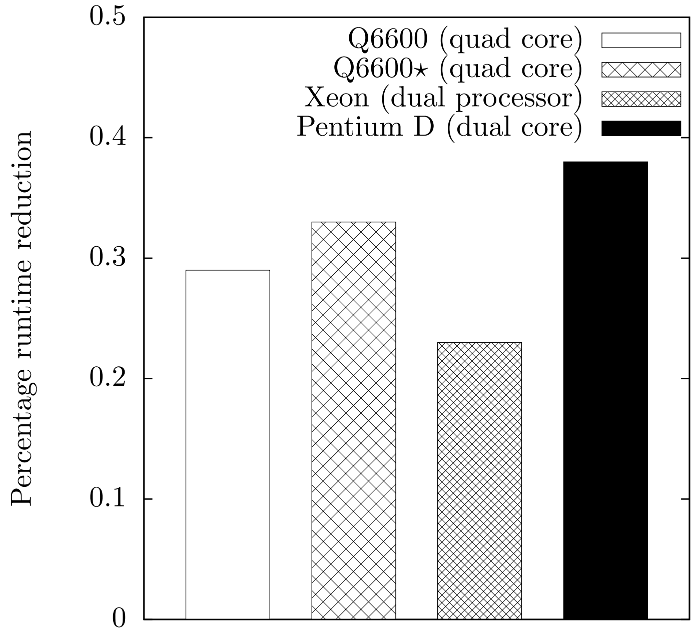


Figure 5.6: Runtime reduction (as a percentage of total runtime) from using periodic parallelisation with four partitions.

tions, taking around 14ms). From this data, spending 20ms per global phase is the 'sweet spot'. More frequent cycling between phases substantially impairs runtime, whilst less frequent cycling brings minimal runtime benefits and increases the risk of the alternative global/local phases distorting the development of the Markov Chain. With the 20ms global phase, the apparent runtime has been reduced by $\sim 30\%$ of the sequential implementation. Whilst falling short of the 45% reduction as predicted by eq. (5.6) when $q_g = 0.4$, $\tau_g = \tau_l$ (as is the case when processing is strictly sequential) and $s = 4$, recall that by restricting the number of partitions to 4 but permitting (requiring) those partitions to be of varying sizes, the size of the largest partition will always be greater than a quarter of the image, and potentially range to the size of the image itself. Consequentially the four processors will never be fully utilised, indeed comparable results can be obtained using only two processors (i.e.

one processor will take the largest partition, the remaining three small partitions performed on the second processor) as demonstrated by fig. 5.6. The differences in performance between the machines is due to the difference between the time per iteration and the overhead required to duplicate, arrange for parallel execution, and merge the partitions. More substantial reductions in runtime can be obtained by using a finer partitioning grid and load balancing if (as in this case) the number of partitions would be greater than the number of available processors. The runtime of the local phase would then tend towards 1/(number of partitions) of the sequential runtime, if the overhead from communication and configuration split/merge operations remained negligible. As mentioned earlier, should the size of each cell in the partition grid become too small then little meaningful work can be done in the $\mathbf{M_l}$ phase as there is no space for features to be moved such that they do not intersect with the partition boundaries and risk interfering with the actions being performed in other neighbouring partitions. A compromise will need to be found on a case-by-case basis, too few partitions underutilise the available hardware whilst too many partitions restrict the work capable of being performed during each $\mathbf{M_l}$ phase.

If implemented using nodes of a cluster instead of pthreads (see page 17) for the processing of local move phase partitions, more moves would need to be performed per phase to compensate for the greater communication overhead. Though this will drive down the rate of convergence (thus increase the total number of MCMC iterations required), the extent to which this occurs will be application specific, as will be the tolerance for errors. The benefits of the greater degree of parallelisation possible will overshadow the additional iterations required until convergence in many applications, particularly in complex images (long iteration consideration times means fewer iterations are required in each phase to overcome the communication overhead, thus a smaller effect on the convergence rate) and/or very large images compared to the average feature size (the more partitions that are possible, the greater the degree of parallelisation and the greater the reduction in runtime).

Although the periodic parallelisation method has been presented so far using two dimensional examples, there is nothing preventing the method from applying to multidimensional spaces. Having partitions of much greater size than the feature size (including the area around a feature that impacts prior or likelihood calculations) is even more important in multidimensional applications as the area in which modifications are permitted becomes a smaller proportion of the total area as the dimensionality increases. For example, in two dimensions if features are 1 unit across and each square in the partition grid is 10×10 units, the area in which the feature centerpoints may be located such they do not intersect a partition boundary is 9^2 units^2, $\frac{9^2}{10^2} = 0.81$ of the total partition area. If in three dimensions partitions are cubes $10 \times 10 \times 10$ and features are still 1 unit across in any dimension the available area for feature centerpoints is only $\frac{9^3}{10^3} = 0.729$ of the total partition area. This is not a problem provided partitions are kept large compared to the size of individual feature and alternating between $\mathbf{M_l}$ and $\mathbf{M_g}$ phases is kept frequent.

5.2 Image Partitioning

Periodic partitioning is a means of allowing moves with local implications to be performed in parallel by making the random proposal of moves less-random over short time frames. If we are willing to make more substantial compromises in exchange for a faster runtime we can partitioning the original image and processing each partition as an independent image (note that this can be applied to higher dimensional spaces as well, but for ease of explanation and visualisation we will consider only two dimensional images for now). For example, consider positioning circles on an image such as fig. 5.7 using the pixel intensity algorithm detailed in section 2.5.2, and that for this image our expected number of circles is 55. If we were to quarter the image and treat each quarter as a entirely independent image with an expected number of circles of $\frac{55}{4} = 13.75$ (we can correctly assume the features are

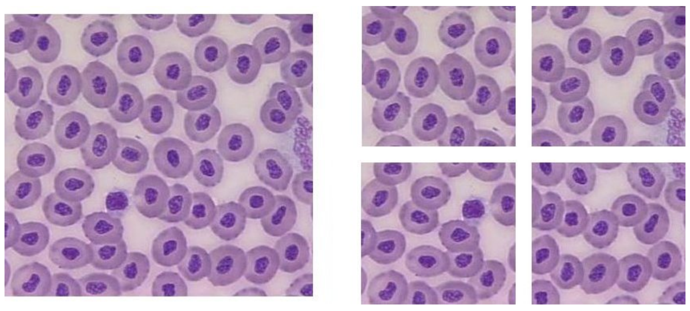


Figure 5.7: Left: an image of immune cells. Right: the same image partitioned into four smaller images. Image partitions have a much smaller statespace than the parent image as both the dimensionality (number of features) and range of values (image area) are reduced.

evenly distributed) how much faster would the simulation run? The runtime of any single iteration depends on how the prior and likelihood values are calculated. If the optimisations detailed in section 2.6 apply (each of the likelihood and/or prior values used in considering a move proposal are derived by calculating the *change* of those values caused by the move, taking constant time rather than time proportional to the total number of features) then reducing the number of features in the model will have little effect on the runtime-per-iteration. If those optimisations are not in play then iterations on the smaller image may proceed substantially faster, depending on the complexity of the prior/likelihood calculations. For instance a reduction to $\frac{1}{4}$ if the prior/likelihood scale linearly, $\frac{1}{16}^{th}$ if the prior/likelihood are of order $O(n^2)$. Additionally, the statespace is much reduced as there are both fewer features and a smaller area in which each feature may be located in the smaller image. It will therefore take fewer iterations until the chain for the small image converges to equilibrium.

Predicting, or even detecting if a chain has converged is unsolved in the general case [15] and beyond the scope of this thesis (as is the determination of the optimum model design and parameters for the rapid convergence to that equilibrium). We will therefore measure and compare program performance using the real-time it takes to propose a state that matches (with some arbitrary degree of accuracy) the target model, this target having been determined by human evaluation of the image, or in the case of autogenerated images from the source configuration from which the image was derived in the first place. For all the following experiments, a chain is considered to have 'converged' when the comparison between the chain's state and the target model meets the following (arbitrary) criteria:

1. 90% of features in the resultant model can be matched to corresponding features in the target model with an error in their position and radius of less than $< 10\%$ of their target radii.

2. The number of remaining, unmatched features make up $< 10\%$ of the expected number of features in the image.

3. The number of features in the resultant model is within 10% of the number of features in the target model.

Performance is be measured by the time taken for these criteria to be met. Since testing these criteria involves additional computation that may vary depending on the circumstances of the test (the number of subimages for instance) obtaining an accurate average runtime is a two stage process. First, the average number of program cycles (the nature of which change from one parallelisation method to another) required to satisfy the matching criteria will be obtained by frequently testing the state of the MCMC chain against the target model. Secondly the MCMC program run will be repeated for that number of program cycles, but with the comparisons tests against the target model omitted. It is only the results of this second stage that will be timed, preventing differences in the number of chain-target model comparisons performed distorting the results. To avoid unnecessary complications with testing for the above criteria we will not use images that contain features that overlap to any significant degree and the relevant parameters in the

Relative area	1	4	16
Expected # features	14	55	220
$\approx$ time per iteration	9×10^{-6}	2.5×10^{-5}	2.3×10^{-4}
$\approx$ iterations to converge	5×10^{3}	5×10^{4}	1×10^{6}
Runtime, sequential (seconds)	0.045	1.25	230
Relative runtime	1	28	5111

Table 5.1: Timing values are taken from runs on a Q6600 processor. The image used for for the relative area = 4 column was figure fig. 5.7(left). The image used for the relative area = 1 column was *one* of the smaller images in fig. 5.7(right)

calculation of the prior term will be set to strongly discourage the generation of models with overlapping features.

In table 5.1 we use this criteria for convergence to compare the use of the section 2.5.2 algorithm on three images of differing size, each smaller image being a subset of all the larger images.

Even with all things being equal, the number of iterations required until 'convergence' is highly variable, and is further modified by the the setup of the simulation: the various parameters for the prior, likelihood, and Metropolis-Hastings calculations and set of model-altering moves and their proposal probabilities. The iteration count and runtime taken for convergence in table 5.1 should therefore be viewed as providing an order-of-magnitude rather than a precise result.

For images similar to that in fig. 5.7, processing time scales with both n (the expected number of features) and A (the area of the image). It is therefore advantageous to consider four images each of area $\frac{A}{4}$ rather than a single image of area A, as each of the smaller images may be processed in parallel. In the above example it is worth partitioning even if multiple processors are not available, as the time to process four smaller images works out a lot shorter than the time needed to process a single larger image (i.e. $4 \times 0.045 \leq 4 \times 1.25 \leq 230$). This suggests the trivial strategy of carving up the initial (large) image into a number of subimages that are then treated as independent images in their own right. The subimages can then be processed in parallel (or not) and the resultant models combined.

Unfortunately in the general case this method cannot be applied without losing the statistical properties that make MCMC attractive in the first place. There are three main issues to consider:

1. Scaling. Assertions that are true on one scale may not hold on smaller scales. For example, the distribution of features may be considered uniform across the entire image, but on the scale of subimages may be clustered. Some subimage may contain many features whilst others contain none. A single expected feature density value inherited from a complete image in which features are not uniformly distributed between the partitions will result in features from different partitions being unevenly matched at best, or many spurious results at worst.

2. Boundary anomalies. Features intersecting the image boundary are prone to cause anomalous results as potential features found in one partition will not interact with potential features from another. A feature may be matched twice (once each side of the boundary), incorrectly, or not at all. Duplicate and mismatched features would need to be reconciled outside of the MCMC method.

3. Model Confidence. One of the key elements to the MCMC method when compared to similar methods is its capability of providing a confidence estimate in the models it produces. Repeated sampling over a long, converged, MCMC chain will yield many possible models. The frequency with which similar models crop up provides a probability for that interpretation of the image. Heuristics added to solve the previous two points may impose biases in the results and invalidate such statistical analysis.

5.3 Intelligent Partitioning

So far we have avoided introducing anomalies as a consequence of parallelisation by parallelising the internals of the MCMC iterations or by overlapped MCMC iterations whilst remaining in the context of a single complete image. However, there is a subset of applications with which we can be more direct in our approach to parallelisation. If the target features are sufficiently disperse and identifiable, it may be possible to find some completely different algorithm to 'pre-process' the image and segment it into sub-images such that features do not intersect (or even approach) the subimage boundaries. Note that features must be far enough away from the subimage boundaries to avoid their presence influencing the results of any neighbouring subimage. Assuming confidence in the segmentation provided by this pre-processor, each partition may now be treated as a independent image that can be processed on a separate thread, processor or machine. Combining the results from each partition is trivial, a simple union of the located features is sufficient as there will be no features crossing or approaching the partition boundaries.

A good pre-processor must be both reliable and fast. The validity of this method hinges on the user having complete confidence that the pre-processor will not create subimages that whose boundaries intersect with any potential features. As for speed, the purpose of partitioning the image is to allow MCMC processing of different parts of the image to take place in parallel. Any improvements in runtime resulting from this parallel processing must take into account the initial time required by the pre-processor to generate the partitions to be performed in parallel, there is no point using intelligent partitioning if the pre-processor uses up all the time that would be gained from parallel-processing the partitions.

Creation or selection of a pre-processor must be done on a case-by-case basis as so much is dependant on the characteristics of the datasets to be processed. In general though, a pre-processor will be a two step process. First some filter will be applied to the image data to identify 'whitespace', areas that are certain to be devoid of interesting features. Next lines (of as simple a form as possible) are drawn through continuous areas of whitespace such that the image is partitioned, hopefully evenly, into many pieces each containing some 'non-whitespace' pixels. Depending on the application this may be some trivial algorithm as in section 5.3.1 below, or some more complex off-the-shelf technology, possibly even a simply MCMC or genetic algorithm. In these latter cases we assume that finding non-feature areas bisecting the image is substantially easier and faster than identifying the details of specific features. Assuming that the image data permits a suitable pre-processor, and that such a processor has been found/constructed, the major difficulty with intelligent partitioning is the derivation of suitable prior assumptions for the partitions (issue item 1). As commented earlier, assumptions that hold on one scale may not apply when restricted to looking at a subset of the image. If the feature density is assumed to be constant throughout an image then the partitions will inherit this value thus assigning potentially inaccurate prior information to some partitions. The degree to which this matters depends on how the likelihood and prior terms are balanced and whether enough iterations will be performed on the most feature-dense partition to allow full convergence.

Ideally the estimate for the properties like the feature density should be mechanically generated based on the actual image data, in which case the same mechanism used to obtain the estimate for the complete image should be applied to the partitions. For example, a good estimate for feature density in certain cell sample images can be obtained by the use of a threshold filter that reduces a greyscale or colour image to binary image. For every pixel (x, y) in the image G we calculate the pixel's colour intensity (as given by the function $I(x, y)$) and test if it is greater than some threshold value ρ i.e.

$$\forall (x, y) \epsilon G : I(x, y) > \rho \tag{5.9}$$

If yes the corresponding pixel in the binary image is coloured black, otherwise it is coloured white. Assuming only the target cells have a colour intensity exceeding

ρ it is now easy to estimate how many circles are in the image, as we count the number of black pixels to give us the area covered by cells, then divide this by the expected area of an average cell. Since the average cell radii r_μ is already assumed to be known (it is required as one of the parameters to calculate the prior term, see section 2.5.1) the expected area of an 'average' cell will be

$$\pi r_\mu^2 \tag{5.10}$$

thus the estimate for the number of cells in the image can be expressed as

$$\frac{|\{\forall(x,y)\epsilon G : I(x,y) > \rho\}|}{\pi r_\mu^2} \tag{5.11}$$

Note that we may use r_μ in this expression as unlike the estimated circle density (λ) the expected circle radii can be assumed constant throughout all partitions (for these images at least).

Using a pre-processor such as eq. (5.11) to establish prior values devolves some of the operations of the original MCMC algorithm to this pre-processor, using the (presumably) faster algorithm to reduce the statespace the Markov Chain will need to explore. Complete confidence in the reliability of the pre-processor is required, thus restricting its use to situations where the presence, or more importantly the *absence* of features can be ascertained. Care must also be taken in the structuring of the MCMC problem, for example if a means to mechanically determine the expected number of features is not available, a pre-processor to crop whitespace from an image is safe if the expected number of features is expressed directly, but not if it is in terms of features per unit area. The reverse holds true if the pre-processor partitions instead of crops, although in either case determining the feature count mechanically as in eq. (5.11) would be preferred. Assuming that the subimage partitioning is correct and that appropriate prior variables can be obtained, the time to obtain a good match for the image can potentially be substantially reduced, yielding MCMC results much faster than any of the parallelisation options explored so far. The degree of parallelisation (and hence performance improvement) of this method is not necessarily under the user's control. Being able to split the image into subimages requires the features of the complete image to be arranged in a convenient way, so the number, size and feature-density of subimages may not be predictable. Consequently the program's runtime for any given image is also unpredictable.

In summary, intelligent partitioning uses some trusted algorithm (the details of which will be application specific) to divide an image into partitions that can be processed independently. Under favourable conditions a large image may be split into multiple small images, each of which can be processed much faster than the original yet with results that can be trivially combined to apply to the original, undivided image. The required conditions are:

1. That the image can be mechanically divided into partitions such that no feature intersects the partition boundary
2. A fast and efficient algorithm exists to conduct this partitioning
3. The features are spread out in the image, and so will not all occur in the same partition (the more evenly spread amongst the partitions the features are the greater the speedup)

Unfavourable images may not offer any parallelisation opportunities at all, and will incur the unmitigated expense of a pre-processor attempting to split the image. This method is therefore highly sensitive to the specific application: the expected distribution of features in the image and the ease by which non-feature areas can be identified. Note that as with periodic partitioning, this method applies not just to two dimensional images but multidimensional areas as well, although more dimension that are present the more challenging the task of partitioning will be.

5.3.1 Example

Figure 5.8 (top-left) shows a number of latex beads in a Petri dish. Due to the clumping of the beads and the ease by which the beads may be distinguished from

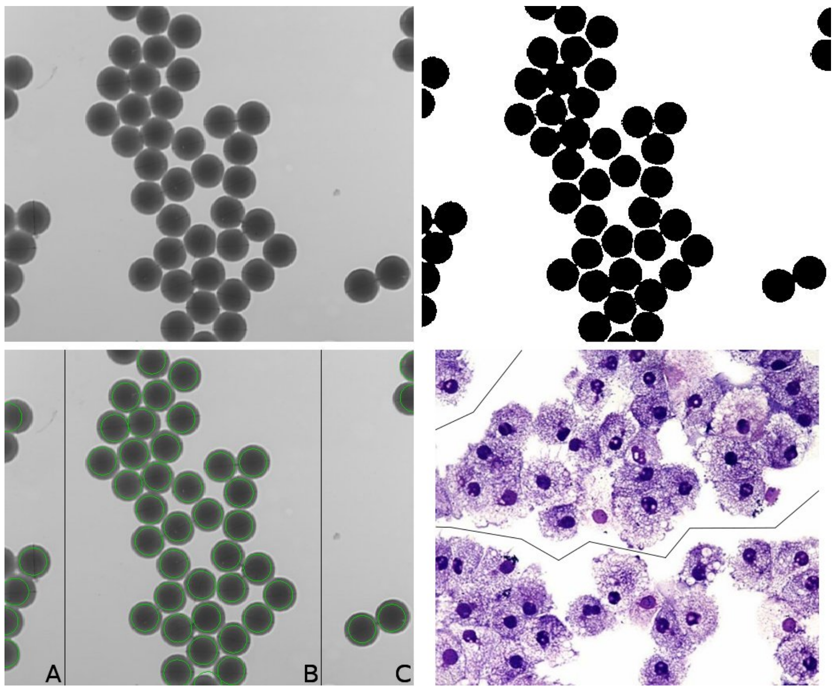


Figure 5.8: Intelligent Partitioning in action. Top left: original image of latex beads in a Petri dish. Top right: threshold filtered image. Bottom left: intelligent partitioned image, post MCMC processing. Bottom right: Intelligent partitioned image of white blood cells

their surroundings this makes a good candidate for the application of intelligent partitioning. We apply a threshold filter as in eq. (5.11) where $\rho = 0.5$ and pixel intensity values in the range 0..1 to identify the likely features (fig. 5.8, top-right), and then partition the image by scanning the filtered image for rows or columns that are completely empty. The partitions are made on columns/rows equidistant between the closest columns/rows containing pixels(s) that passed the threshold criteria (fig. 5.8 bottom-left), the entire partitioning procedure taking negligible time compared to the subsequent MCMC processing. From applying eq. (5.11) to this image we know that there are 48 features. Were this figure only provided as part of the *prior* knowledge, we might be forced to assume the distribution of features

		A	B	C
Area (pixels^2)	2.13×10^5	3.14×10^4	1.33×10^5	4.82×10^4
Relative area	1	0.147	0.624	0.226
# features (visual)	48	6	38	4
# features (constant den.)	–	7.08	29.97	10.86
# features (thresholding)	46	4.9	38	3.1
$\approx$ time per iteration	4×10^{-5}	1.9×10^{-5}	4.3×10^{-5}	2.0×10^{-5}
$\approx$ iterations to converge	27 000	4 000	22 500	900
Runtime (seconds)	1.08	0.08	0.97	0.02
Relative runtime	1	0.07	0.90	0.02

Table 5.2: Results of intelligent partitioning on fig. 5.8. Timing values are taken from runs on a Q6600 processor. Number of iterations and runtime required averaged over 20 runs.

was uniform, thus allocating an expected feature count of 7 to partition A, 30 to partition B and 11 to the partition C. Fortunately this image contains no elements that can be confused with actual features when using the threshold filter, so we may calculate an estimate for number of features (λ) based on eq. (5.11). The results using this information are in table 5.2: If at least 3 processors are available the intelligent-partitioning program runtime is the longest time taken to process any of the partitions (in this case 0.97), as combining the results for the three separate partitions is trivial. With only two processors load balancing should be used, which for this example gives the same runtime of 0.97 (as $0.07 + 0.02 < 0.97$).

An irregular partitioning as in fig. 5.8 (bottom right) imposes little additional overhead on the MCMC algorithm once the partition lines have been drawn. The likelihood and prior calculations will be oblivious to the partitioning as the pixel data for neighbouring partitions will be blanked out (this is safe to do as the validity of intelligently chosen partitions depends upon the presumption that the contents of neighbouring partitions are irrelevant to the consideration of the current partition). Since there will be no pixel data for beyond the partition boundary, features will not be placed there by the MCMC algorithm. Should additional checks be necessary to keep all features within the partition bounds, they will take place when changes to the model are proposed, before the prior and likelihood calculation (that dominate runtime) take place. The only difficulty will be the creation of such irregular partition boundaries and the time this takes, though since detecting where features definitely do not exist is easier than identifying with certainty the position and properties of features, a range of comparatively fast segmentation algorithms (some of which may themselves be based upon MCMC) will generally be available.

5.4 Aggressive Partitioning

So far we have avoided posing any substantial threat to the statistical validity of parallelisation. Speculative moves and chains are purely implementation-based measures that do not affect the MCMC algorithm, periodic partitioning attempts to sidestep problems with the statistics, and intelligent partitioning uses some other algorithm to safely partition the image. There are some applications where MCMC is a convenient means of structuring a solution to some problem, but the statistical robustness offered by MCMC is not strictly required - obtaining a ‘reasonable’ answer promptly is more important than waiting for a statistically pure result. In this section we address methods which do just that. To be clear, unlike the methods discussed in previous chapters the following methods are *not* statistically equivalent to conventional MCMC and so are not guaranteed to eventually converge on the target solution. Though it depends on the application, these may produce ‘reasonable’ solutions albeit with the potential for anomalies and biases for certain type of configurations. They are not expected to explore multiple interpretations, and should be considered only if an MCMC-like method is required to obtain an approximate solution.

5.4.1 Blind Partitioning

The intelligent partitioning method described in section 5.3 uses some application specific algorithm to partition an image then treats each partition as a completely separate image. Certain criteria (depending on the specifics of the application) have to be met in order for suitable prior variables for each partition to be derived from those of the original image. For example, the determination of the number of cells in an image using eq. (5.11) requires the cells to be of greater intensity than all other features in the image. Aside from the partition-prior criteria, intelligent partitioning also relies upon the existence of an efficient partitioning pre-processor, and for features in the image to be sufficiently dispersed as to allow meaningful partitioning. To obtain the full benefits of the partitioning, the partitions must be of approximately equal size and each contain approximately the same number of features each. These restrictions severely limit where intelligent partitioning may be employed, for example the features in fig. 5.7 are too densely packed and there is insufficient differentiation between target and non-target pixels for useful partitions to be made.

A simple adjustment to this method is to do away with the partitioning pre-processor and simply partition the image in some arbitrary manner, without worrying about bisecting potential features (thus either greatly simplifying the requirements of the partitioning pre-processor, or doing away with it altogether). When the time comes to combine the results for each partition we can employ some heuristics to attempt to procedurally 'patch up' any anomalies resulting from the partitioning. To do this in a systematic fashion whilst minimising anomalies we propose there be overlap between each partition such that the largest expected feature will fit entirely inside, as in fig. 5.9 (top left). In this figure the thick dotted line represents the conventional partition boundary, whilst the thin dotted line marks the actually boundary used when the partition does MCMC processing, fig. 5.9 (top right). When merging the resultant configurations, features with their centerpoint in the non-overlapping regions are automatically accepted, whereas features with centres in a overlapping region will need comparing with nearby features from the other partition(s) - fig. 5.9 (top right). If the MCMC algorithm applied to the partitions yielded good results, such features should appear in both partitions with minimal differences and so can be merged with little difficulty - fig. 5.9 (bottom left). Features without a counterpart from the other partitions are disputable, you may wish to accept or discard them depending on whether it is more important to avoid false-positives or not missing potential features. Either way, this method is more consistent, reliable and less context sensitive than intelligent partitioning (which may fail to partition in useful manner, i.e. if all features are grouped in one place). It is also competitive in terms of overhead, the merging of the partition results is a deterministic processes that considers each feature in turn and (at worst) compares it with all the other features from each of the partitions' results (a $O(n^2)$ process with the number of features across all the partitions). The time this takes is negligible compared to the time required to run perform the many thousands of MCMC iterations to produce the results in each partition.

The trade-off for the speed and simple implementation of blind partitioning is the reduction in confidence in the final result. Unlike intelligent partitioning and non-partitioned MCMC the statistical assurances accompanying MCMC cannot be applied to the final model produced for the image. Anomalies may occur along the partition boundaries if neighbouring partitions do not agree on results in the overlap between partitions. Anomalies may also occur in the interiors of the partitions if the prior values applied to the partitions are not 'accurate'. Though this method will work well when the MCMC simulation reliably finds a single interpretation for each feature in every partition, should multiple interpretations for a feature arise in an overlap region anomalies (false-positives, unmatched features) are practically guaranteed as there will not be a simple way of deciding which interpretation is correct (if indeed it is possible to detect that the features from the multiple partitions

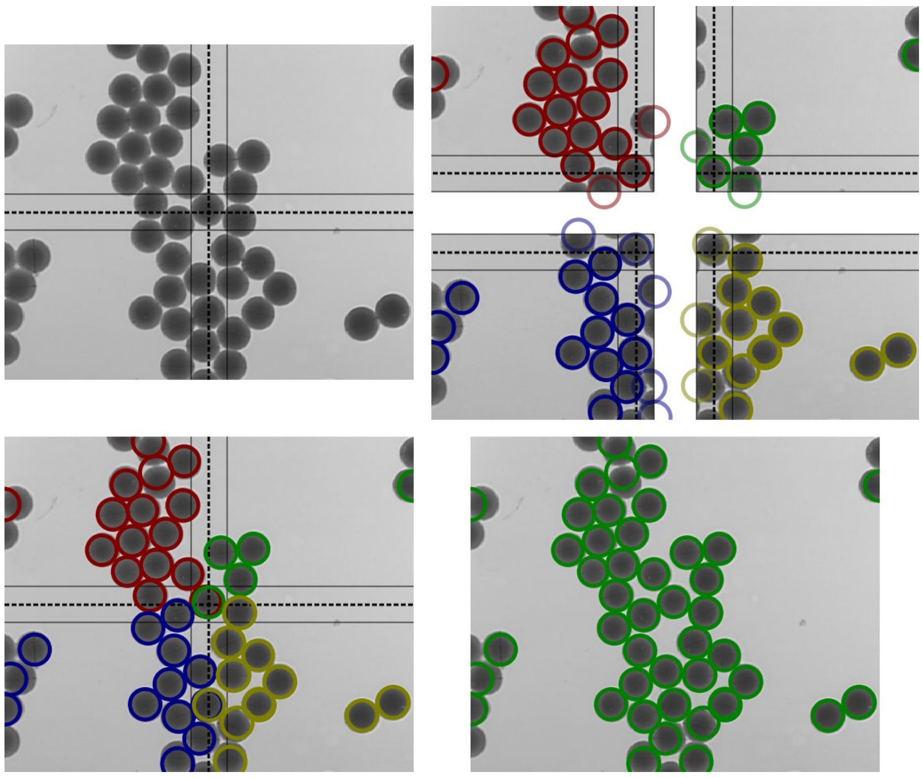


Figure 5.9: Blind Partitioning in action. Top left: original image of latex beads in a Petri dish, with partition boundaries marked. Top right: Image partitions after MCMC processing. Bottom left: Trimmed partition models overlaid. Bottom right: Merged model for the image.

refer to the same entity in the original image). Of course, the use of partitioning and the recombination heuristic make it impossible to obtain the differing model alternatives along with their relative probabilities, of of the unique benefits of normal MCMC.

The benefits of blind partitioning thus depend on the specifics of the application being processed. Under favourable conditions (features are unambiguous, prior information is determined procedurally from image data) blind partitioning may result in dramatic reductions in runtime whilst still producing results free of any obvious anomalies from the partitioning. In unfavourable conditions the run-time benefits may be less impressive (if workload does not end up split between the partitions/processors) or with the presence of many anomalies (if prior information for the partitions was incorrect, or if features were ambiguous and open to multiple interpretations).

Example

Using the same example as in section 5.3.1, the image is first split into four equal sized areas as shown by the dotted lines in fig. 5.9, top left. These areas are then expanded to the solid lines to fully enclose any feature whose centre was in the original area. In this case we have extended each partition boundary edge by 1.1 times the expected circle radius, easily encompassing such features as there is very little variation in the radii of the latex beads. After determining the expected number of circles in each partition using eq. (5.11) the MCMC algorithm was applied to give the results in top right of fig. 5.9. Circles not completely enclosed in the partition (features whose centre is not inside the dotted line marking the simple quartering of the image) are deleted from each partition's model. The union of the partition's models is then taken and any circles centred in the overlap area that are in close proximity (centerpoints within say 5 pixels of each other) are merged (replaced with a circle with centerpoint and radii are the average of the original circles).

A comparison between processing the complete image and the partitions is given in table 5.3. The runtime of the whole procedure (if four processors are available) is approximately equal to the longest time taken to process a partition as the merging of the partition models is takes negligible time compared to thousands of MCMC iterations. In the example the runtime was reduced to 27% of the original, with no apparent anomalies present as a result of the partitioning. Note that there is an uneven distribution of features between partitions, thus workload between the

		A	B	C	D
Area (pixels^2)	2.13×10^5	6.42×10^4	6.42×10^4	6.42×10^4	6.42×10^4
Relative area	1	0.3	0.3	0.3	0.3
# features (visual)	48	18	7	17	14
# features (constant den.)	–	14	14	14	14
# features (thresholding)	46	18.0	7.02	17.9	15.3
$\approx$ time/iter	4×10^{-5}	2.844×10^{-5}	2.52×10^{-5}	2.97×10^{-5}	2.89×10^{-5}
$\approx$ iters	27 000	4 600	3 400	9 600	4 100
Runtime (seconds)	1.08	0.13	0.09	0.29	0.12
Relative runtime	1	0.12	0.08	0.27	0.11

Table 5.3: Results of blind partitioning as shown in fig. 5.9. Timing values are taken from runs on a Q6600 processor. Number of iterations and runtime required averaged over 20 runs.

processors. Images where features are more evenly distributed may be processed faster, whereas images where features are heavily clustered may not obtain as much benefit from blind partitioning.

5.4.2 Approximating the Initial Model

It can be noted that while the MCMC method requires a initial model, in tests so far we have always set this as a random configuration of features. In principle this model may be chosen from anywhere in the statespace. The closer the initial model is to the optimum state (the state with the maximum posterior probability), the faster the simulation can be said to have converged (though unless the simulation is left to run for long enough to conduct a more complete examination of the statespace, this convergence may well be to a local not global optima). If we can quickly generate a good guess for the initial model we can shave a substantial amount of time off the runtime. Even an initial model with obvious flaws will probably prove better than starting from a random configuration. How then, should we obtain such a guess? One option is to use the blind partitioning method to obtain an initial model, and then run conventional MCMC on the result to clear out any remaining anomalies. The difficulty here is that we have no idea how may conventional MCMC iterations will be required, although a solution to this would need to be found anyway, even for the conventional application of MCMC (detecting convergence in the general case, and predicting how long before a MCMC simulation converges are unsolved problems beyond the scope of this thesis). In practise this will require experimentation and experience ("it takes X iterations to properly match image A, B, and C, so it will probably take the X iterations to match the similar image D"). The more subtle difficulty is that if blind parallelisation imposes a strong bias on how the configuration is formed, we may not run the final MCMC chain for long enough for it to escape this biased interpretation. There will certainly be the temptation to stop the simulation 'early' since it will be quickly producing 'reasonable' models for the image (just not necessarily the optimum). The use of $(MC)^3$, specifically intended for aiding the escape of local optima (see section 2.4.3) is recommended.

5.5 Summary

This chapter has explored how the Markov Chain Monte Carlo algorithm can be adapted to process large images by treating them (temporarily or not) as containing a collection of smaller images that may be processed more-or-less independently. The methods examined fall into two categories: 1) those that will (theoretically) have a negligible effect on the end-result of the simulation and 2) those that will produce a result that is 'good enough' for some applications but lack statistically backing or guarantees. If there are many features in an image, considering only a subset of the image at a time results two reductions in statespace complexity: each individual feature will be restricted to a much smaller area and there will be fewer features to consider at a time. The time saving is potentially huge, but best results require effective and reliable pre-processors and/or post-processors specific to the application.

The partitioning techniques detailed in this chapter have been expressed in terms of a two-dimensional application, however the methods also apply to applications of three dimensions and higher. Whilst the overhead costs for partitioning and merging may differ (e.g. detecting collisions between features in three-dimensional space is more expensive than in two dimensions), the end result of halving the statespace and considering each half separately will be the same. The only significant difference between two and three dimensional implementations is the added complexity/difficulty in constructing an efficient pre-processor for the intelligent moves.

Chapter 6

Conclusions and Future Work

6.1 Supplementing Existing Parallelisation

Speculative moves, chains, and even periodic partitioning may all be utilised in conjunction with each other and existing MCMC parallelisation schemes, such as $(\mathrm{MC})^3$ (see section 2.4.3). For any scheme involving multiple chains, one simply replaces the single processor machines performing each chain with a multicore/multiprocessor machine, and implements speculative moves (and optionally speculative chains) on each chain. In the case of $(\mathrm{MC})^3$ the cold chain will get the most benefit from the speculative moves, as the 'hot' chains are more accepting of move proposals therefore will trigger the use of their speculative move less frequently. The net effect of combining speculative moves/chains with $(\mathrm{MC})^3$ is hard to predict or evaluate as it depends on the importance of the hot chains in improving mixing and the exploration of the statespace, but if speculative moves/chains reduce the runtime over a sequential implementation, bringing speculative execution to $(\mathrm{MC})^3$ can only improve the rate at which a solution is obtained. Periodic parallelisation can also be combined with $(\mathrm{MC})^3$, either using multiple machines in the cluster to process each partition in each chain, or just keeping one machine per chain and using multithreading to process the work of the various partitions.

Speculative moves, chains, and periodic parallelisation also complement the embarrassingly parallel view of Markov Chain parallelisation (section 2.4.1) as they function to reduce the time until convergence, traditionally the prohibitive feature of Markov Chain Monte Carlo. Once convergence has been reached on each of the chains the remaining runtime for acquiring a set number of samples scales normally with the number of chains. The more direct forms of image splitting (intelligent and blind splitting) are best viewed as a means to simplify the problem prior to proper MCMC processing. Given the substantial benefits of dealing with small subsets of a large image, such a method should be used if at all possible when dealing with models involving large number of features, if the loss of statistical certainty of eventual convergence on the optimal solution is acceptable.

6.2 Guidance for Implementers

When developing an MCMC algorithm, readers are advised to incorporate the parallelisation methods described in this document into their planning and designs from the outset, as some of the parallelisation and optimisation techniques place constraints on the MCMC implementation. (for example, it must be possible to calculate what the prior and likelihood of a state would be after some proposed move is applied to it, without actually applying the move or making any other modification to the existing state - see page 64). Whilst image-splitting based techniques may easily be adapted to a MCMC implementation through the addition of pre and post processing operations, speculative moves and speculative chains have very specific requirements. For instance, it must be possible to treat potential moves as discrete objects whose processing and/or application may be deferred or prevented, and that can be evaluated without any modification (even temporary) of the model representing the MCMC chain's state.

To avoid future scientists reinventing the wheel, as the example applications featured in this document were constructed and tested the parallelisation, core

MCMC mechanics, boilerplate, performance measuring and recording functions were abstracted out into a extensible framework called pMCMC, which is documented in appendix A. pMCMC automates the implementation of the parallel MCMC methods described in chapters 3 and 4 and some of those in chapter 5. It also provides (through static code and autogeneration) most input/output functionality that is required for a basic MCMC application, including for the handling of all the prior and likelihood that are needed for determining the posterior probability, the move proposal probabilities, and any 'tweak' values that can be used to fine-tune the execution of the parallelisation. This allows developers using the pMCMC framework to focus on the specifics of the MCMC algorithm relating to the program's application - the specification of the simulation state and the calculation of the prior and likelihood probabilities. Appendix B shows an example implementation of one of the section 2.5 circle-finding algorithms and appendix C demonstrates the runtime usage of this application.

When faced with either implementing a new MCMC application or adapting an existing one, the first matter to consider is whether some form of image splitting is applicable to the application and datasets that are expected to be processed, and whether image splitting is acceptable considering the purpose to which the resultant models will be put. Depending on the form of image splitting employed and the information that can be gathered about the input data before employing MCMC, image splitting may or may not introduce anomalies or result in uneven matching throughout the dataset. In almost all cases, image splitting will not result in statistically 'pure' results though there is the potential for substantial reductions in runtime, even with a limited number of processors. In many cases the only way to determine if the potential loss of reliability and accuracy is worth the reduced runtime is to try it and see.

When it comes to the actual MCMC code, there will be considerable variation in the types and behaviours of various MCMC simulations, and different parallelisation methods will yield significantly different results depending on the properties of the simulation and hardware in question. If the MCMC algorithm has been implemented using the pMCMC framework, determining the appropriate parallel processing methods to employ is simply a matter of trying each of the parallelisation options in turn and observing the results (the parallel settings to use are specified either on the command line or in the XML `job` file defining the simulation to run). If pMCMC is not being used, a selective approach to which parallelisation methods are implemented is desirable to avoid unnecessary work. Analyse the workings of each move type and determine the relative time spend in the prior calculations, likelihood calculations, and move proposal construction/application. If the bulk of the time is spent in the prior and likelihood calculations and the time for the prior calculation is $\approx$ the time for the likelihood calculation, consider performing the calculations for the prior and likelihood in parallel as this can halve the time spent calculating them. Unless a high proportion of iterations are accepted, speculative moves will provide a reliable and predictable performance improvement (see section 3.3 for best-case estimates based upon move rejection rate). Speculative chains should be implemented only if there are move types whose proposals involve substantially more processing time than the other move types (see section 4.4 for an idea of how significant the differences in move processing time should be before considering speculative chains). Periodic parallelisation will yield good results so long as there are have large datasets containing relatively small features that can be modified with localised moves. Unlike image splitting, periodic parallelisation is statistically sound (given sufficiently frequent phase switching) but in general slower than the direct image splitting methods. If image splitting has been deemed acceptable, employing periodic parallelisation as well would be redundant.

6.3 Thesis Summary

The work described in this thesis has been concerned with the parallelisation of a class of algorithms known as Markov Chain Monte Carlo. The statistical basis for this type of algorithm prevents the simple application of traditional parallelisation methods, and most existing multiprocessor implementations originate from the statistics discipline with the intent to improve the rate of chain convergence. This thesis address the problem from the high performance systems discipline, and seeks to spread the computational burden across multiple processors and/or machines.

To concisely summarise the contributions of this thesis:

- Two new methods have been presented that allow Markov Chain Monte Carlo algorithms to take advantage of multi-core and multi-processor machines. Termed 'speculative moves' and 'speculative chains', they are described and evaluated in chapters 3 and 4 respectively. Being purely implementation-based changes, the end results are unaffected whilst the runtime of typical MCMC programs can be reduced by $\sim 40\%$ by using just two processes.

- A new modification of Markov Chain Monte Carlo termed 'periodic parallelisation' has been proposed, that permits partial parallel processing on a large scale with a limited (and statistically acceptable) impact on the results. This method is covered in section 5.1.

- A number of methods have been considered that can reduce the runtime of MCMC applications concerned with image processing problems by considering portions of the image (temporarily or permanently) as independent images in their own right. Whilst lacking the statical certainty accompanying the other parallelisation methods presented, the potential runtime improvements are substantially higher whilst giving results that will be reasonable for many applications. These methods are explored in sections 5.2 to 5.4.

- The new methods proposed in this thesis have been fully implemented on a number of different machine architectures, and the suitability of these architectures for these new approaches demonstrated and compared. The practical results and comparisons can be found in the chapters concerning each of the methods.

- A means of predicting the runtime of MCMC programs using our speculative moves (section 3.3), speculative chains (section 4.3.1) and periodic partitioning methods (section 5.1.1) have been constructed. The remaining methods presented in this thesis have had practical examples conducted demonstrating the typical runtime improvements that can be expected (see section 5.2 and sections 5.3.1 and 5.4.1). This provides: (i) increased certainty in real-world MCMC applications, (ii) a means of comparing alternative supporting architectures in terms of value for money and/or performance.

- A programming framework that automates much of the construction of MCMC programs has been developed. The parallelisation methods of speculative moves, speculative chains and periodic parallelisation can automatically made available with no extra work necessary from the the implementer. The usage of this framework is described and demonstrated in the appendices.

After the introductory chapter described the layout of this thesis, its primary contributions, and introduced the terminology used throughout, chapter 2 presented the background research relevant to the contributions of this thesis. This included an explanation of the Markov Chain Monte Carlo method and a discussion of how and where it may be applied. A summary of the existing methods of improving MCMC using parallel processing was presented with examples, along with an explanation of the conventional means of parallelising the MCMC algorithm and how these methods differ from the novel methods presented in this thesis. The chapter went on to establish a specific context for the work presented in the rest of the thesis by describing in detail two MCMC applications for the segmentation of circular formations in a bitmap image, and in doing so further explaining the details of the most general purpose form of the MCMC algorithm (the Metropolis-Hastings transition kernel). Some simple non-parallel optimisations were also covered here for the benefit of readers implementing their own MCMC application. The example applications shown here also served as the testbed for the parallelisation methods presented in later chapters.

Having provided background and context to MCMC applications in chapter 2, chapter 3 presented the first contribution of this thesis, the parallelisation method 'speculative moves'. Once the rational for this method and the revised MCMC implementation were explained, a formula for calculating the predicted runtime whilst using speculative moves was constructed. The speculative moves method was then tested on the practical example programs presented in chapter 2 using a number of different hardware platforms, and these results are compared with those predicted from the mathematical formula.

The logical development of speculative moves, termed 'speculative chains', was covered in chapter 4. Since a mathematical formula describing the benefits of this method quickly becomes unmanageably complex when attempting to describe anything but the simplest situations, a simulator was constructed and used to predict the runtimes that can be obtained using speculative chains. As with speculative moves, speculative chains was tested on the practical examples from chapter 2 using a number of hardware platforms.

Periodic parallelisation and a variety of image-splitting methods were presented in chapter 5. Unlike speculative moves and chains, the methods presented in chapter 5 modify the basic MCMC algorithm in ways that are not be appropriate for all applications. However, with suitable applications, careful implementation and thorough testing these parallelisation methods can produce a substantially larger reduction in runtime that either speculative moves or chains would be capable of.

The software developed for this thesis consists of a framework with which to construct MCMC applications quickly and efficiently, without the implementer needing to write repetitive boilerplate code. Applications constructed with this framework (termed pMCMC) can implement the three major and new parallelisation methods presented by this thesis with minimal work from the application implementers. An overview of the pMCMC framework and its benefits to any MCMC implementers occupies appendix A. To demonstrate the ease by which fully-featured MCMC applications may be developed using this new framework constructed for this thesis, appendix B contains an example implementation using pMCMC on one of the circle-finding methods from section 2.5. Finally the usability of the applications built with pMCMC is shown in appendix C, where an example of how end-users interact with a pMCMC program at runtime is provided.

6.4 Limitations and Future Work

Although the results from chapters 3 to 5 show clear benefits from the various methods, the exact improvements obtained will depend on the specific characteristics of the application to which they are applied. Potential factors that may impact the final runtime include:

1. The time per iteration
2. The time per phase of each iteration (i.e. time to calculate the prior, time to calculate the likelihood)
3. The move rejection rate
4. The proportion of different types of move ($\mathbf{M_s}$, $\mathbf{M_f}$, $\mathbf{M_g}$, $\mathbf{M_l}$ etc).
5. The performance characteristics of different parts of the program, dependant on the hardware, operating system, and compiler/compiler settings utilised.

6. Caching issues, with SMP parallelisation methods conflicting simultaneous memory access request could cause problems, potentially memory thrashing.

Further exploration of the consequences of changing these factors on an MCMC program's runtime is desirable, particularly with the aim of creating or improving means of predicting runtimes (either though formulae or fast simulators such as from section 4.3.1). To date all such factors have been treated as constant over a simulation run, more accurate runtime predictions may be possible if the consequences of varying these factors at runtime are examined. For example, if starting with a underpopulated initial state the time per iteration would likely increase as more features are located and the size of the model (and statespace) increases. The move rejection rates are also very likely to change as the simulation nears convergence (a 'near perfect' state is going to reject more moves than a random initial state), the benefits of the speculative methods therefore change throughout the simulation.

The simulations used to predict results in chapters 3 and 4 can be refined to more accurately reflect the implementation. For example, the simulator from section 4.3.1 should be updated to account for non-negligible duration state-cloning. Predictions for specific applications and platforms would also be enhanced by considering the varying costs of mutex operations over different hardware and software systems.

Another issue not fully addressed is determining the most appropriate size of partitions when using periodic partitioning (section 5.1). The smaller the partitions the greater the benefits from the parallel processing in $\mathbf{M_l}$ phases (assuming sufficient processors), however a smaller partition size also means a greater proportion of the image is unable to be modified during that $\mathbf{M_l}$ phase (as features that intersect with any of the partition boundaries may not be modified, and no modifications can be proposed that would cause a feature to intersect with a partition boundary), potentially delaying convergence of the chain. There is also the option of using speculative moves on the work done in each partition, as an alternative to shrinking the size of each partition. It would therefore be useful to determine the extent to which these concerns and options conflict in some typical MCMC applications, and how to arrive at an optimum compromise.

One matter particularly relating to the image splitting methods of chapter 5 is that of load balancing. There is no guarantee that subimages will require equal amounts of processor time to process, in which case the order in which subimages are scheduled for processing may greatly effect the final runtime. Load balancing is also a concern for all methods in heterogeneous multiprocessor environments (such as in clusters with processors of different capabilities). Task scheduling in such an environment is an active research area, one which both the implementation and predictions of all methods from chapter 5 could benefit.

Since the efforts here have been split between constructing the pMCMC framework, implementing the parallelisation methods and implementing and fine-tuning the programs for the section 2.5 algorithm, it was not possible to implement and test a wider range of MCMC algorithms and applications. With the pMCMC now providing parallelisation and support code it is hoped that implementers with a specific expertise in MCMC algorithms will construct pMCMC applications of greater complexity and scope than was possible for this thesis. It would also be interesting to examine how different the differing memory footprints and access patterns impact the runtime of the various parallelisation methods. Another area for exploration is determining the extent to which more traditional MCMC programs (integral approximation being the classic example) can benefit from speculative moves and chains.

Finally there are opportunities for expanding the pMCMC framework. Currently the framework provides `Move` classes implementing move operations for models consisting only of an unordered collection of independent features. A set of classes to facilitate the use of models with inter-feature relationships (features organised in binary trees, for instance) would be helpful for would-be-developers for applications that exhibit non-independent features. Support for distributed execution across mediums other than MPI may be useful, and a more functional user interface would make the end-user simulators more accessible. More important would be in-built support for additional MCMC variants, $(\mathrm{MC})^3$ for instance. Due to the internal design of pMCMC adding periodic parallelisation-like variants such as $(\mathrm{MC})^3$ should be a relatively easy undertaking.

6.5 Concluding Remarks

This work has sought to bring together the disciplines of statistics, image processing and high performance systems. A number of parallelisation strategies have been devised and tested, most of which will compliment existing methods targeted at improving the rate of simulation convergence. Although the methods that are applicable and the performance improvements that can be obtained will vary depending on the particulars of the specific application and implementation, the estimates for the best possible reductions in runtime along with real-world examples obtained (using a variety of hardware architectures and representing a number of potential MCMC application characteristics) provide a good indication of whether a particular application will benefit. In some cases a fair estimation of the improvements can be obtained.

It is hoped that the pMCMC lowers the barrier of entry for would-be MCMC implementers, and facilitates more research into MCMC methods and applications. It is the intent that the parallelisation options provided with pMCMC will allow the practical implementation of more end-user MCMC applications by making infeasible and uneconomical methods feasible, and improve the efficiency of those MCMC applications already discovered.

Appendix A

The pMCMC Framework

Though not the main focus of this thesis, in the process of implementing and evaluating the aforementioned parallelisation methods a framework for the rapid development of MCMC applications was developed: pMCMC. An example implementation using this framework is given in appendix B and the runtime usage shown in appendix C. The purpose of this framework is to separate the task of constructing and tuning an MCMC simulation for a specific application from the task of implementing the parallelisation methods presented in this thesis. As a side-effect, the implementation of many tedious and/or error-prone aspects of an MCMC application have also been automated, including but not limited to user interaction (via XML and a variety of frontends), sanity checking, the Metropolis-Hastings kernel and generic aspects of the MCMC algorithm.

In section A.1 we briefly describe the MCMC method and its uses. Section A.2 gives an overview on how the pMCMC framework is used to create an MCMC application. Section A.3 describes some of the internal design structure and decisions that were made. Whilst for the most part users of pMCMC will not interact with these components, a commentary on them may be of interest to those considering extending pMCMC or writing their own MCMC program from scratch. In Section A.4 we show how the applications generated using pMCMC are used, and in section A.5 we look at the overhead involved in the use of this framework.

A.1 Introduction

In its most basic form an MCMC algorithm is simple to implement, as demonstrated by the following psuedocode:

```
do {
    ProposedMove p = makeProposal();
    double mh_value = metropolisHastings(p);
    if (random()<mh_value)
      apply(p);
    else
      abort(p);
while(!done);
```

Transitioning from the seemingly straightforward sequential implementation to one or more of the parallel implementation described in chapters 3 to 5 can be a daunting task to those not accustomed to parallel programming. Extensive rewrites may be necessary if the transition is not planned for from the outset. For instance, speculative moves requires move proposals be created and evaluated without any changes to chain's state, prohibiting a mechanism of applying then rolling back proposed changes should the proposal be rejected (a viable sequential implementation that has been encountered). Similarly, speculative chains demands the existence of secondary (speculative) states to exist and be developed by MCMC iterations before being potentially merged with the primary state, and both speculative chains and periodic parallelisation require proposed moves to be suggested from a subset of the possible move types depending on the phase of the simulation (separating $\mathbf{M_s}$ and $\mathbf{M_f}$ for instance).

The programming knowledge and experience required for the technical implementation of parallel processing (`pthreads` [48], `MPI` [24, 30] and safe parallel programming practises) will not necessarily be found by the initial developers of a MCMC method (whose experience will be focused on statistical algorithms and image processing). Additionally if one is developing a number of MCMC applications there is substantial repetition of effort and the writing of tedious and repetitive boilerplate code (i.e. for selecting between move types with the correct probabilities and generating the suitable proposed moves for the move type, based on probabilities and other parameters specified by the user is some fashion). The pMCMC framework was created to address these issues and provide a convenient testbed for the rapid testing of the parallelisation methods developed for this thesis.

Creation of the the code for performance monitoring, input/output of data and simulation properties and multithreading instructions serves as a further distraction from the MCMC implementor's primary focus: the simulation's model, the possible model transitions, and the efficient calculation of the posterior probability. To combat these problems and to make the creation of parallel MCMC applications more accessible to the theorists the pMCMC has been developed. Through a combination of a library of source files, templating and automatic code generation a specific MCMC application can be plugged into a generic parallel MCMC kernel in a manner that allows the programmer to focus purely on the application specific components of the MCMC application.

Functionality automatically provided by the pMCMC framework includes:

- The implementation of the Metropolis-Hastings transition kernel.
- The speculative moves and speculative chains implemented using `pthreads`, and the periodic parallelisation mechanism implemented using MPI.
- Multiple executables for different situations: for testing, for SMP machine execution, and for MPI execution.
- Use of XML files for configuring all the MCMC simulation variables ('prior' values, move proposal probabilities etc).

- Automatic generation of much ‘boilerplate’ and housekeeping code (for instance the random selection of a type of move to execute based upon the proposal proabilities provided via an XML `job` file, and the tracking of simulation statistics such as the average acceptance probability for each type of move).
- Recording of simulation metrics (timing of individual steps and the program as a whole, actual move acceptance rate).
- Optional XML logs of the MCMC simulation’s setup and statistics gathered during the simulation’s execution.
- Programmatic interface for integration with your own frontend (an optional OpenGL display is available).

On a Q6600 the pMCMC framework is capable of performing up to 3.2 million iterations per second whilst in sequential mode. Parallel processing performance is highly dependant on the specific characteristics of the application, but in practical tests using just one of the parallelisation methods available (speculative moves) allowed for up 40% reduction in runtime just by using a dual-core or dual-processor system, with no additional coding required compared to that for simple sequential execution.

A.2 Component Overview

The pMCMC framework greatly simplifies the creation of parallel MCMC applications by providing an implementation of the generic MCMC code with parallel processing support already provided. In order to implement this with the minimum of performance overhead the application specific code must be integrated into the pMCMC code using a combination of templating, `typedefs` and automatic code generation. The application specific code the developer must implement are:

- A description of the simulation from which automatic code generation can take place.
- The `Collection` describing the simulation model (its ‘state’ at any one time).
- A set of potential move proposal generators, one for each ‘type’ of move that may be made to the simulation model.
- The means to calculate the prior and likelihood for a given simulation model, and from a simulation model and move proposal taken together.

An overview of what is involved for the developer using pMCMC follows.

A.2.1 Defining the Simulation

The application-specific characteristics of the simulation must be provided in a suitable format to allow autogeneration of boilerplate and integration code. We have used GNU Autogen[*] as the code generation program, available in most Linux distributions. Technically autogeneration is not required, but handcoding the required files is repetitive, tedious and error-prone. Autogeneration is not required at runtime, and as long as the user-specified definitions are not changed the files do not need to be regenerated at each compile.

An example of an `autogen` descriptor file is given in section B.1, this shows the definition file used for the circle finding algorithm in section 2.5.1. The `prior` and `likelihood` entries list the runtime-specific terms used to calculate the prior and likelihood terms in the MCMC acceptance test. The runtime values of these properties will be specified by the user at runtime (typically via an XML file) and are made available to the developer's code via autogenerated and manged `Settings` objects.

The `move` entries describe the moves by which the simulation may advance. If a move type requires additional user-input to define how it should operate (for

[*]`http://www.gnu.org/software/autogen/`

instance, the proportion by which a property may be modified in a single move) these are detailed in associated `moveproperty` elements. In addition to announcing the presence of each type of move there is also logical information (such as the move type that may reverse the move, used in sanity checking the move proposal probabilities), display information (for presenting the end-user with meaningful information), and optimisation information (if a move does not alter the prior term, there is no point recalculating the prior term).

A.2.2 The Model

The model class (termed `Configuration` in the pMCMC framework) describes the 'state' of the Markov chain simulation. The construction of a suitable configuration to describe structures in the input image is the purpose of performing the MCMC simulation. A `Configuration` class must be a C++ `Collection`, preferably a `Vector` or `List`. Each object in this collection should represent a feature of the the simulation model, a feature representing a node or branch in a graph or a specific discrete structure in the image, such as a circle). Whilst a unordered list of features has been assumed to date in pMCMC development, composite features may be created by establishing links/references between feature objects. The model class must also implement methods to establish the value of the likelihood and prior terms for any given configuration (an example of the method signatures these classes should implement is given in sections B.2 and B.3).

If it is appropriate and desirable to see a (2D) visual representation of the model (useful to confirming the simulation is working correctly and to aid when using trial-an-error to fine-tune the simulation for a reasonable convergence rate) a `DrawableFeature` class should be provided to serve as a 'buffer' for the information to be drawn. Objects of this drawable class must be capable of storing the displayable properties of a `Feature` class, and drawing a representation of those properties to an OpenGL area on demand (see section A.4 and section A.4.3). The method signatures required for such a class can be seen in section B.4

A.2.3 The Moves

Many of the bookkeeping and boilerplate code associated with the moves will already have been defined in the `autogen` definitions file from section A.2.1, all that remains is the code to generate and apply the changes associated with each move. This is done by implementing a single method with a set name and signature for each of the potential move types, the names of these methods being determined by the names of the moves as set in the `autogen` definitions file (see section B.5 for an example of this). These move implementation methods must return a `Proposal` object that holds all the information concerning a yet-to-be applied move, can calculate the effect the application of the proposal would have on the existing simulation state, and effect that state change if the proposal is accepted. Whilst superfluous to a sequential implementation, this design is required for the speculative move/chain methodologies where multiple proposed moves must be considered simultaneously then selectively applied.

The pMCMC framework has been developed and tested with MCMC models that consist of an unordered list of independent features. Multiple implementations of the `Proposal` interface for the possible types of alterations that can be made to such a model are available to implementers, all that needs to be done is to provide the specifics of how the model can be changed. Preimplemented `Proposal` classes have been written assuming that the log/prior terms for a configuration are simply the product of the terms for each of the features in said configuration. The available types of move supported are BIRTH, DEATH, SPLIT, MERGE and REPLACE. The 'replace' move type encompasses any alteration made to a single feature (as far as the prior/likelihood terms are concerned, the feature is being removed and a new feature added with one or more properties different, all in a single iteration). Moves that may have a knock on effect on the prior probability density must be flagged as such in the definitions file (and thus force complete recalculation of that value for the entire configuration when a change is made), otherwise the prior and likelihood values will be determined by the change in value that will be caused by the application of the move.

If the simulation's model contains features arranged in some form of order or heirachy (for instance, each 'feature' is considered a branch or node of a tree) then appropriate model-modifying code must be supplied. Otherwise the provided `Proposal` subclasses can be used, all that is required is the calculation of the new properties for the feature(s) that will be changed.

A.3 Internal Design Considerations

To support both sequential and parallel transition kernels efficiently each aspect of the MCMC simulation needs to be maintained as distinct components. Potential changes to the simulation's state are represented as objects of a `Proposal` class that are capable of determining the prior and likelihood of the new state being proposed without changing (or duplicating, even temporarily) the existing state of the simulation. Speculative moves are implemented by creating and considering `Proposal` objects on different `pthreads` simultaneously. The maintenance of chain statistics, and the implementation of the transition kernel (in both speculative and non-speculative forms) is all held within a `Chain` object.

The instructions on what work to perform, a combination of the simulation settings, parallelisation options, initial simulation and the image data are all contained in a `Job` object, which can be marshalled to/demashalled from an XML file (termed a `Job` file). It is the responsibility of a `Runner` object to take a `Job` and perform the required action on a `Chain`. The `Runner` also handles interspacing MCMC iterations with timing, monitoring, and user update actions. The state of the simulation as reported to the user (either through a progress bar or an OpenGL display of the simulation's state) is buffered by the `Runner`. This is to allow the display of information to the user to be refreshed or redrawn at any time as required by the frontend, without unnecessarily delaying the execution of the simulation[*].

Partitioning-based parallelisation is implemented by having different implementations of the `Runner` class. Whilst the basic version simply ensures the user is kept informed of the state of the simulation, when partitioning-based parallelisation is required a MetaRunner is used that partitions the original `Job` and passes the subset jobs to either conventional `Runner` objects or `RemoteRunner` objects, `RemoteRunner`s being 'handles' to a normal `Runner` class executing on a separate machine. The existing implementation only contains a `MPIRuner` implementation of the `RemoteRunner` interface, using a communication channel other than MPI is simply a matter of implementing an alternative `RemoteRunner` to handle sending and receiving the sending of `Jobs`. Since `Job` objects can be (un)marshalled to/from XML, and little additional information need be exchanged between the `MetaRunner` and the remove 'slave' `Runner`s to implement the partitioning parallelisation mechanisms, adding support for additional inter-machine communication channels is a relatively simple matter.

Other classes of note are `MoveSet` and `MoveType`. A `MoveType` object contains information about a specific type of move (for instance, alter position). Not only how such a move should be used but also potentially useful statistics concerning those moves that were used, such as the number that were ultimately accepted/rejected. The `MoveSet` class is responsible for randomly selecting the appropriate `MoveType` to use to generate a new `Proposal`. In the simplest case this is selecting with the move proposal probabilities defined in the `Job` file. This task becomes more complicated when dealing with speculative moves (chapter 3), speculative chains (chapter 4) or periodic parallelisation (section 5.1) as the selection is then from a subset of the available moves, such as randomly choosing between only those `MoveType`s classified as having short processing times, or only those considered to have localised effects.

[*]Aside from the consumption of resources used by the GUI thread

With the configuration, moves, and prior/likelihood calculations implemented all that remains is to compile and link your code and the pMCMC's code together. The default build system used is CMake*, the build system used for KDE 4. Alternative build systems (handwrite Makefile, GNU Automake/Autotools) may easily be written.

A.4 Using the Simulator

Currently the pMCMC framework produces three different executables, with names derived from a single root name. If the MCMC simulation was entitled `foo` these generated programs will be:

`gfoo` This program implements a simple graphical frontend (using GLUT†) to display the state of the simulation as it runs. Real-time control of the simulation is limited to start, stop and to step through the simulation one move or simulation cycle at a time (useful for debugging). If a more advanced frontend is required it can be developed on-top of the existing structure, see section A.4.3. This program is generally used for testing or demonstration purposes, to confirm that the MCMC simulation is constructing a suitable model.

`cfoo` This variant is command-line only, and intended for running in sequential mode or on SMP machines. By default it is configured for minimal user interaction to facilitate the fastest possible execution in scripts or via work schedulers (such as PBS‡, SGE§ or Condor [56]). Progress information can be displayed on the command line on request.

*`http://www.cmake.org`

†A quick and easy library for displaying OpenGL images, see `http://www.opengl.org/resources/libraries/glut/` and `http://freeglut.sourceforge.net/`

‡`www.openpbs.org`

§`http://www.sun.com/software/sge`

`mfoo` The MPI version of the program, primarily used for periodic parallelisation. Again, runtime user-interaction is kept to a minimum.

Statically compiled variants of these programs (e.g. `cfoo-static`) may also be created if required libraries will not be available on the machines to which the program will be deployed. In all program variants the progress of the simulation can (optionally) be monitored by a progress bar on the command line, 'progress' being defined either in terms of the number of iterations performed or by the results of some compile-time declared fitness function that is evaluated at regular intervals. Once execution terminates, statistics concerning the simulation will be displayed (again, optional) and/or written to a log file for further analysis.

In typically usage work the description of the simulation to run is supplied in a `job` file, along with command line options instructing what parallelisation options should be employed. The output will be a file containing the final configuration and an optional `log` file describing the simulation and containing statistics concerning its execution. Optionally the program(s) may be set to provide 'snapshot' files of the simulation's state at regular intervals, either for debugging purposes or in the regular usage of MCMC. As described in section 2.2.3, MCMC is often used to explore multiple potential solutions through the regular sampling of a simulation that has reached equilibrium.

A.4.1 Jobs and Logs

Work is submitted to these programs via a `job` file that describes the simulation variables and how the program should run the simulation. A 'template' `job` file is created by the program for the user to modify to meet their needs. When run, each program will (optionally) record all the simulation and program parameters in a log file, along with the initial and final state of the MCMC model and statistics concerning how the simulation progressed. To facilitate ease of integration with existing and future tools, both the `job` and `log` files are XML files.

An example `job` file is shown in appendix C. The end user of the MCMC program need only understand how to edit such files to customise the MCMC simulation (all program variants can generate example 'template' files for the user to modify). As for the developer, the bulk of the XML reader/writer code is autogenerated from the Autogen definition file from section A.2.1). The only XML reader/writer code the developer need write is that for the model description (used in elements `initial_model` and `final_model`, and content of these elements need not even be XML. In keeping with the ethos of accessibility and standardisation, SVG*, the W3 standard for vector graphics, was used as the model descriptor for the implementations of the section 2.5 algorithms. Each circle (the features be identified) being represented as a `svg:circle` element.

This same XML schema is also used as the wire protocol for communicating instructions between physically different machines (in the current implemented, using MPI to setup and communicate between nodes/machines). The speculative moves and speculative chains methods are implemented only in shared memory environments, those parallel methods that may be applied to a cluster of machines consist of MCMC chain(s) running for an arbitrary length of time (or number of iterations) thus the overhead of interpreting to and from XML will be made to be negligible. The precise form by which model information is structured (i.e. the content of the `initial_model` and `final_model` elements) is not specified by the framework, whilst XML is recommended for consistance and clarity (should the output files and logs be manually inspected) a more concise format that can be processed more rapidly can be utilised at the application implementors discresion.

The log files are very similar to `job` files, but have an additional set of elements describing the MCMC simulation that was performed. At a minimum, information such as the number of iterations performed, the number of each type of move were accepted/rejected, etc. If detailed monitoring was enabled the informa-

*`http://www.w3.org/Graphics/SVG`

tion gathered will also be recorded in the log file.

A.4.2 Detailed Monitoring

The pMCMC framework is capable of monitoring and recording many statistics concerning simulation runs. For instance, the mean and standard deviation in the time it takes each type of move to execute, the number of each type of move that are accepted and rejected, and the time it takes each step in the simulation cycle to execute[*]. Since the collection of this information imposes an overhead that may occupy a significant proportion of program execution in some circumstances, advanced and detailed simulation monitoring is disabled by default and must be explicitly enabled to be used.

In order to facilitate debugging and the like, extensive reports off many aspects of the program execution may be recorded, either to a standard output channel or a file[†]. The amount of logging and debugging level is set along a scale ranging from NONE, ERROR, WARNING, INFO, DEBUG. Error and Warning log messages refer to failures to correctly process some aspect of execution, and must be dealt with by the MCMC developer. INFO level debugging records key events and echos the modification of internal settings (to double check the correct data is being used by the simulation). The DEBUG logging level records very detailed information, including the results of the move acceptance test(s) conducted at each MCMC iteration. Obviously enabling logging to this level will severely hurt program performance. For this reason by default only ERROR and WARNING messages are enabled.

[*]The exact definition of 'step' here depends on the parallelisation mechanism(s) in use. In sequential MCMC one simulation step is exactly one MCMC iteration, whereas with speculative moves enabled one step involves one attempted MCMC iteration on each thread that is available, potentially equalling multiple MCMC iterations.

[†]For full logging functional the rlog[‡] library is required, if this is unavailable to commandline and log file is still possible, but may be less efficient.

Developers using pMCMC may also make use of this logging mechanism by the use of simple macros of the form

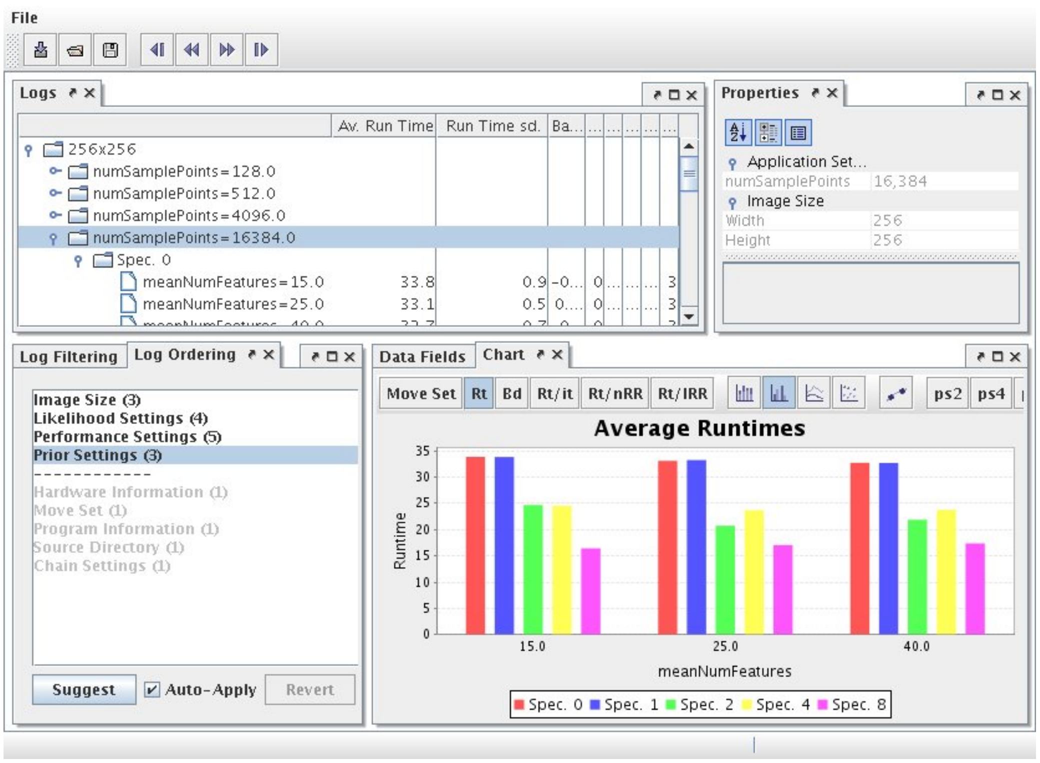
```
logWarning(("Warning text, using printf style arguments, \%s",
                "For example, this string"));
```

These instructions may be removed entirely by the setting of a single compile-time flag if there are any concerns of the logging infrastructure inserting unacceptable overhead.

A.4.3 Frontend API

When the `gfoo` frontend is insufficient a more advanced frontend is possible using the pMCMC programmatic interface. Whilst data and simulation configuration input must still be provided as XML (either a file or a text block in memory), the programmatic interface does allow the simulation to be started, stopped, and stepped through, and for various data to be fed back to the application frontend. The visual representation of the simulation's state (its current model) as seen from the `gfoo` program variant can be displayed simply by providing an appropriate OpenGL area, the display will then be configured and written on. To limit the extent the frontend limits processing speed, requests for information (either statistics, or an update of the OpenGL display of the simulation) take place only at fixed intervals that may be set through the programmatic interface or on the command line when the program is initiated. By setting a low update/refresh rate on the displayed information the adverse impact of any GUI frontend can be kept to a minimum.

Although not currently implemented, the same code autogeneration mechanisms used for creating the backend may be utilised to automatically generate a suitable frontend editor/view for the job files. That the pMCMC and its API are written in C++ and use the CMake build system make KDE the obvious choice, although any frontend capable of calling C++ code will be sufficient.



```
Logs                                   Av. Run Time  Run Time sd.  Ba...
256x256
   numSamplePoints=128.0
   numSamplePoints=512.0
   numSamplePoints=4096.0
   numSamplePoints=16384.0
      Spec. 0
         meanNumFeatures=15.0         33.8          0.9  -0...  0 ... ... ... 3
         meanNumFeatures=25.0         33.1          0.5  0....  0 ... ... ... 3
```

```
Properties
Application Set...
numSamplePoints  16,384
Image Size
Width            256
Height           256
```

```
Log Ordering
Image Size (3)
Likelihood Settings (4)
Performance Settings (5)
Prior Settings (3)
------------
Hardware Information (1)
Move Set (1)
Program Information (1)
Source Directory (1)
Chain Settings (1)
```

Figure A.1: The Log File Analyser in operation

A.4.4 Results Analysis

To aid in the rapid analysis of data generated when testing the parallelisation methods described in this document a Java program has been developed to read in large numbers of log files, group them according to the differences in runtime variables and average data obtained from repeated runs with identical initial conditions (except for the initial state). Figure A.1 shows this program in operation. The bottom-left panel ('log ordering') lists those parameters that are not constant throughout the examined set of log files. The top-left ('logs') shows the the results arranged in a tree, each level of the tree corresponds to an aspect that is different between the log files whilst each branch represents a distinct value that differs. The ordering of these differences in the logs panel is determined by the order of the list in log-ordering panel. Differences in those properties listed below the dotted line in the difference list of the log-ordering panel are ignored in the log panel, the results for all such tests are collated into a single node. Examining a node reveals the average properties (such as runtimes) obtained from all log files contained within that node, information that is displayed in the top-right panel ('properties').

The bottom-right ('chart') panel shows a graph of the runtime results for a selected node. If the selected type of graph supports it, the data presented will be classified into groups as represented by direct children and grandchildren. In this manner complicated graphs can be generated with just a few mouse clicks.

A.5 Case Studies

All experimental results pertaining to speculative moves, chains, periodic parallelisation and image splitting were generated from applications created by the pMCMC framework. To determine the maximum potential of applications implemented using this framework, a baseline application was quickly developed that implements all methods required for the compilation and execution of a pMCMC application whilst doing as little actual work as possible (for instance, the 'model' was a vector containing only one element that was never changed, only one move was possible and that did nothing). This baseline application could perform $\sim$ 3 200 000 sequential iterations per second on the Q6600. Compare this to the rate of processing for the test applications (from 2.5): when configured to look for a mere five circles a rate of only $\sim$ 175 000 sequential iterations per second could be achieved. The overhead imposed by the pMCMC itself can therefore be considered negligible for any serious application, though the overhead imposed by actually multithreading (through the pMCMC or some handcoded implmenentation) must be considered against the benefits of that multithreading on a case-by-case basis.

A.6 Conclusions

The pMCMC framework is a fast and easy way of developing MCMC simulations. The developer need only implement the application specific aspects of the MCMC simulation. Once correctly implemented, a selection of parallelisation methods are available for free, as is the majority of code for file input/output, user interactions, monitoring and recording. The resultant applications will automatically record logs of all simulations performed. A Java base log file analyser is available to aid in determining the settings and parallelisation methods necessary for the most efficient rate of processing.

By freeing the MCMC developer from user interaction and parallel processing code and allowing them to focus on the specifics of their MCMC algorithm we hope to make efficient MCMC solutions more accessible and promote development of more ambitious MCMC projects by alleviating their main downside - the slow rate of execution and the difficulty in combatting this by parallel processing.

Appendix B

Example Implementation using pMCMC

Here we show the definition file and header files used to implement the circle finding algorithm presented in section 2.5.1. A brief overview of how the definition file and class and method implementations combine to create a fully functional program is given in appendix A, but for a complete understanding of how this is performed, refer to the pMCMC manual. Our intent in providing these listings is to demonstrate how little work is required to obtain a working set of pMCMC applications.

The definitions file `AppSpecificSettings.def`, used by the automatic code generator is provided in full. For brevity we will omit the C++ implementation code and just list the class and method signatures found in the header files. The `Cell` class describes a single feature in the image, whilst `CellConfiguration` describes the model for the entire image. `DrawableCell` contains the instructions needed to draw a `Cell` in OpenGL. `MoveSetImpl.cpp` contains the implementation code for the available move types, here we only list the method signatures not the full implementation.

B.1 AppSpecificSettings.def

```
AutoGen Definitions AppSpecificSettings.tpl;

prog-name        = cells;
prog-title       = "Cell Identification Program";
version          = "1.0";
home-page        = "http://dcs.warwick.ac.uk/~jbyrd/";

settings-name = "CellSettings";
xml-element-name = "app-settings";
namespace = "cells";
xml-name = "cells";
supplimentary-includes = "#include <cmath>";

Feature-Class = "Cell";
Configuration-Class = "CellConfiguration";

/*************************************************************
 * Prior, likelihood and derived values
 *************************************************************/

constant = { name = "sqrt_2pi";
             type = "double";
             value = "std::sqrt(M_PI*2.0)";
             desc = "The square root of 2*PI"; };

prior = { name            = "radiusMean";
          default         = 20;
          min             = 1;
          max             = 1000;
          display-precision = 1;
          readable-name = "Radius Mean";
          desc            = "The mean radius the cells are "
                            "expected to take"; };
prior = { name            = "radiusStdDev";
          default         = 2;
          min             = 0;
          max             = 1000;
          display-precision = 3;
          readable-name = "Radius std. dev.";
          desc            = "The expected standard deviation from "
                            "the mean radius"; };
```

```
prior = { name              = "meanNumFeatures";
          default           = 15;
          min               = 1;
          max               = 100000;
          display-precision = 1;
          readable-name     = "Mean␣num.␣features";
          desc              = "The␣mean␣number␣of␣features␣expected␣"
                              "in␣the␣image"; };
prior = { name              = "overlapDensityFactor";
          default           = "5";
          min               = 0;
          max               = 200;
          display-precision = 2;
          readable-name     = "Cell␣Overlap␣Density␣Factor";
          desc              = ""; };

likelihood = { name              = "numSamplePoints";
               type              = "unsigned␣int";
               default           = 64;
               min               = 4;
               max               = "32768";
               display-precision = 0;
               readable-name     = "Num.␣Sample␣Points";
               desc              = "The␣number␣of␣points␣around␣a␣"
    "proposed␣cell␣to␣examine␣to␣determine␣the␣its␣likelihood.";};

derived = { name          = "radiusVariance";
          source          = "radiusStdDev";
          readable-name   = "Radius␣Variance";
          setter          = "radiusVarianceValue␣=␣"
                            "radiusStdDevValue*radiusStdDevValue;";
          desc            = "The␣expected␣variance␣from␣the␣mean␣"
                            "radius"; };
derived = { name          = "expNMeanNumFeatures";
          source          = "meanNumFeatures";
          readable-name   = "e^(-Mean␣num.␣features)";
          setter          = "expNMeanNumFeaturesValue␣=␣"
                            "std::exp(-meanNumFeaturesValue);";
          desc            = "<i>e<i>␣to␣the␣power␣of␣the␣negetive␣"
                            "mean␣number␣of␣cells"; };
derived = { name          = "samplePoints";
          type            = "double*";
          init            = "NULL";
```

```
            destruct      = "if(samplePointsValue!=NULL) { "
    "delete samplePointsValue; samplePointsValue=NULL; }";
            source        = "numSamplePoints";
            readable-name = "Sample Points";
            desc          = "Array of the coordinates at which to "
                            "sample a unit circle.";
            setter        = <<- STR_END
  if (samplePointsValue != NULL) {
    delete samplePointsValue;
  }
  samplePointsValue = new double[numSamplePointsValue*2];

  double angleFraction = 2*M_PI/numSamplePointsValue;
  for(unsigned int i=0; i<numSamplePointsValue; i++) {
    samplePointsValue[i*2]     = cos(i*angleFraction); // x coord
    samplePointsValue[(i*2)+1] = sin(i*angleFraction); // y coord
  }
STR_END; };

/************************************************************
 * Moves
 ************************************************************/
move = {    name          = "add";
            reverse       = "delete";
            alters-prior  = "true";
            localized     = "false";
            slow-move     = "false";
            readable-name = "Add";
            jacobian      = "1";
            desc          = "Add a single feature to the "
                            "configuration";
            req-conf-size = "1"; };
move = {    name          = "delete";
            reverse       = "add";
            alters-prior  = "true";
            localized     = "false";
            slow-move     = "false";
            readable-name = "Delete";
            jacobian      = "1";
            desc          = "Delete a single feature from the "
                            "configuration";
            req-conf-size = "2"; };
move = {    name          = "merge";
```

```
            reverse       = "split";
            alters-prior  = "true";
            localized     = "false";
            slow-move     = "false";
            jacobian      = "0.125";
            readable-name = "Merge";
            desc          = "Replaces two features in the "
     "configuration with a single feature consisting of averaged "
     "values from the two original features";
            req-conf-size = "2"; };
move = {    name          = "split";
            reverse       = "merge";
            alters-prior  = "true";
            localized     = "false";
            slow-move     = "false";
            jacobian      = "8";
            readable-name = "Split";
            desc          = "Replaces a feature in the "
     "configuration with two new features that, if merged, would "
     "result in the original feature";
            req-conf-size = "1"; };
move = {    name          = "alterRad";
            readable-name = "Alter Radius";
            alters-prior  = "true";
            localized     = "true";
            slow-move     = "false";
            jacobian      = "1";
            desc          = "Modifies the radius of a single "
                            "feature in the configuration";
            req-conf-size = "1"; };
move = {    name          = "alterPos";
            readable-name = "Alter Position";
            alters-prior  = "true";
            localized     = "true";
            slow-move     = "false";
            jacobian      = "1";
            desc          = "Modifies the x and y coordinates of a "
                            "single feature in the configuration";
            req-conf-size = "1"; };
move = {    name          = "longReplacePos";
            readable-name = "Slow Replace Position";
            alters-prior  = "true";
            localized     = "false";
            slow-move     = "true";
            jacobian      = "1";
            desc          = "Randomly repositions a circle to "
                            "anywhere in the image, using a time-"
                            "consuming method";
            req-conf-size = "1";  };

moveproperty = { name    = "alterPosStdDev";
            movename      = "alterPos";
            default       = "2";
            readable-name = "Alter Position Std. Dev.";
            desc          = "The standard deviation by which a "
     "feature's position may be altered in alter-position moves";};
moveproperty = { name    = "alterRadStdDev";
            movename      = "alterRad";
            default       = "1";
            readable-name = "Alter Radius Std. Dev.";
            desc          = "The standard deviation by which a "
     "feature's radius may be altered in alter-radius moves";  };

moveproperty = { name    = "splitRadiusStdDevProportion";
            movename      = "split";
            default       = "0.08";
            readable-name = "Split Move Radius Proportion";
            desc          = "The proportion of the original cell's "
    "radius to be used as the standard deviation when generating "
    "the radii of the new cells in a <i>split feature move</i>";};

/*************************************************************
 * Supplimentary functions
 *************************************************************/
function = {
            name          = "locationDensity";
            returnType    = "double";
            declare       = "locationDensity() const";
            code          = <<- STR_END
  return uniformLocationDensity();
STR_END;  };

function = {
            name          = "sizeDensity";
            returnType    = "double";
            declare       = "sizeDensity(const double rad) const";
```

```
          code          = <<- STR_END
  const double rd = radius-radiusMeanValue;
  return    exp(-(rd*rd)/(2*radiusVarianceValue))
          / (radiusStdDevValue*sqrt_2pi);
STR_END; };
```

B.2 `Cell.h`

```
class Cell {
private:
  double rad;       /**< The cell's radius */
  double xCoord;    /**< The x-coordinate of the cell's center */
  double yCoord;    /**< The y-coordinate of the cell's center */

public:
  Cell(const double x=0, const double y=0, const double radius=0);
  void operator=(const DrawableCell& cell);
  bool operator==(const Cell& cell) const;
  bool operator!=(const Cell& cell) const;
  friend std::ostream& operator<<(std::ostream& s, const Cell& c);
  std::string getBriefDescription() const;
  std::string toString() const;
  void setCoord(const double x, const double y);
  void getCoord(double *x, double *y) const;
  void setRadius(double radius);
  double x() const;
  double y() const;
  double radius() const;
  double radiusSqr() const;
  bool overlaps(const Cell &cell) const;
  bool overlaps(const int x,
                const int y,
                const unsigned int width,
                const unsigned int height) const;
  bool overlaps(const int x,
                const int y) const;
  double areaOverlap(const Cell &cell) const;
  double dist(const Cell &cell) const;
  double distSqr(const Cell &cell) const;
  double dist(    const double x,
                  const double y) const;
  double distSqr(const double x,
                 const double y) const;
  bool matches(const Cell& c,const mcmc::Settings&) const;
};
```

B.3 CellConfiguration.h

```
class CellConfiguration : public std::vector<Cell> {
public:
  CellConfiguration() : std::vector<Cell>() {}
  ~CellConfiguration();

  void trim(const int x,
            const int y,
            const unsigned int width,
            const unsigned int height,
            CellConfiguration& dest);

  static CellConfiguration* read (xmlTextReaderPtr reader,
                                  unsigned int* width,
                                  unsigned int* height);
  void write(xmlTextWriterPtr writer,
             const mcmc::RawImage& rawImage) const;

  std::string save(const std::string filename,
              const std::string initialModel,
              const unsigned int iters,
              const mcmc::RawImage& rawImage) const;
  static CellConfiguration* load(const std::string filename,
                                 unsigned int* width,
                                 unsigned int* height);
  void draw(GLfloat* pixels,const int width, const int height);
  static CellConfiguration makeRandom(
          const AppSpecificSettings& settings,
          mcmc::random1_rng* generator);

  double prior(const AppSpecificSettings& settings) const;
  double logPrior     (const AppSpecificSettings& settings) const;

  double logLikelihood(const ProcessedImage& pImage,
                       const AppSpecificSettings& settings) const;
  double likelihood    (const ProcessedImage& pImage,
                       const AppSpecificSettings& settings) const;
  static double likelihood (const Cell& cell,
                            const ProcessedImage& pImage,
                              const AppSpecificSettings& settings);
  static double logLikelihood(const Cell& cell,
                              const ProcessedImage& pImage,
                              const AppSpecificSettings& settings);

  template<class CellIterator> static double
  logLikelihood(const CellIterator begin,
                const CellIterator end,
                const ProcessedImage& pImage,
                const AppSpecificSettings& settings);
  }

  iterator pickRandom(mcmc::randoml_rng*);

  template<class CellIterator> static double
  prior(const CellIterator begin,
        const CellIterator end,
        const AppSpecificSettings& settings);

  template<class CellIterator> static double
  logPrior(const CellIterator begin,
           const CellIterator end,
           const AppSpecificSettings& settings);

};
```

B.4 DrawableCell.h

```
class DrawableCell {
private:
  static unsigned int numLines; // # line segments in a 'circle'
  double xCoord;
  double yCoord;
  double rad;

public:
  DrawableCell(const double x=0, const double y=0,
               const double radius=0);
  DrawableCell(const Cell& cell);

  void operator=(const Cell& cell) {
    xCoord = circle.x();
    yCoord = circle.y();
    rad = circle.radius();
  }
  void set(double x, double y, double radius);

  double x() const;
  double y() const;
  double radius() const;

  void draw(void) const;
  static void draw(const Cell& c);
  void drawFull(void) const;
  static void drawFull(const Cell& c);
  static void setNumLines(const unsigned int num);
  static unsigned int getNumLines(void);
};
```

B.5 MoveSetImpl.cpp

```
mcmc::ProposedMove*
MoveSet::create_addMove(CellConfiguration& conf,
                        const mcmc::ProcessedImage& pImage,
                        const mcmc::Settings& settings,
                        mcmc::MoveType& moveType,
                        randoml_rng* generator) {...}

mcmc::ProposedMove*
MoveSet::create_deleteMove(CellConfiguration& conf,
                           const mcmc::ProcessedImage& pImage,
                           const mcmc::Settings& settings,
                           mcmc::MoveType& moveType,
                           randoml_rng* generator) {...}

mcmc::ProposedMove*
MoveSet::create_mergeMove(CellConfiguration& conf,
                          const mcmc::ProcessedImage& pImage,
                          const mcmc::Settings& settings,
                          mcmc::MoveType& moveType,
                          randoml_rng* generator) {...}

mcmc::ProposedMove*
MoveSet::create_splitMove(CellConfiguration& conf,
                          const mcmc::ProcessedImage& pImage,
                          const mcmc::Settings& settings,
                                mcmc::MoveType& moveType,
                                randoml_rng* generator) {...}
mcmc::ProposedMove*
MoveSet::create_alterRadMove(CellConfiguration& conf,
                                    const mcmc::ProcessedImage& pImage,
                                    const mcmc::Settings& settings,
                                    mcmc::MoveType& moveType,
                                    randoml_rng* generator) {...}

mcmc::ProposedMove*
MoveSet::create_alterPosMove(CellConfiguration& conf,
                                    const mcmc::ProcessedImage& pImage,
                                    const mcmc::Settings& settings,
                                    mcmc::MoveType& moveType,
                                    randoml_rng* generator) {...}
```

Appendix C

Example runtime use of pMCMC programs

The following XML file contains all the runtime information required to perform the image processing shown in fig. C.1 using the MCMC algorithms presented in section 2.5 and whose pMCMC implementation was outlined in appendix B. To perform 10,000 iterations on the image `white-cells.jpg` using a randomly generated initial model and whilst using speculative moves the following command would be used:

```
ccells --job=cells.job --image=cells.jpg -ocelljob01 -s4 -n10000
```

As a consequence of the above command two files will be generated, `celljob01.svg` containing the description of the final state of the simulation (in SVG format, see section A.4.1) and `celljob01.log` holding information and statistics concerning the simulation. The simulation will use speculative moves with a maximum of four moves being considered simultaneously. Section C.2 shows the (optional) command line output from running the `ccells` or `gcells` program.

C.1 `cells.job`

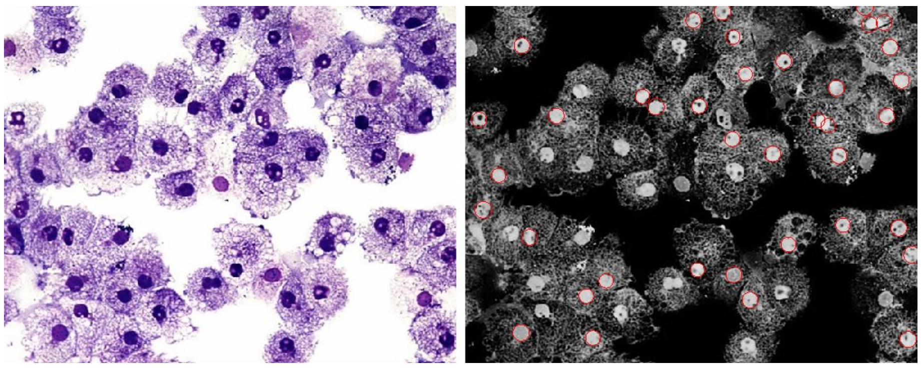


Figure C.1: An image of a collection of white blood cells before (left) and after 100 000 iterations by the program from appendix B using the `job` file in section C.1

```
<job>
 <execute iterations="100000"/>
 <chain>
  <prior_exponent>1.0</prior_exponent>
  <likelihood_exponent>4.5</likelihood_exponent>
  <posterior_exponent>20.0</posterior_exponent>
  <use_cross_correlation_as_likelihood value="false"/>
 </chain>
 <app-settings>
  <likelihood>
   <setd name="numSamplePoints" value="32"/>
  </likelihood>
  <prior>
   <setd name="radiusMean" value="12"/>
   <setd name="radiusStdDev" value="2"/>
   <setd name="meanNumFeatures" value="55"/>
   <setd name="overlapDensityFactor" value="10.0"/>
  </prior>
  <moves>
   <move name="add" prob="0.05"/>
   <move name="delete" prob="0.05"/>
   <move name="merge" prob="0.05"/>
   <move name="split" prob="0.05"
          splitRadiusStdDevProportion="0.08"/>
   <move name="longAdd" prob="0.0"/>
   <move name="longDelete" prob="0.0"/>
   <move name="alterRad" prob="0.3" alterRadStdDev="2.000000"/>
```

```
    <move name="alterPos" prob="0.4" alterPosStdDev="3.000000"/>
    <move name="replaceRad" prob="0.0"/>
    <move name="replacePos" prob="0.0"/>
    <move name="longReplacePos" prob="0.1"/>
  </moves>
 </app-settings>
</job>
```

C.2 Sample Output

```
======================================================================
Proposal Probabilities
======================================================================
  Add                  : 0.05, jacobian=1,     alters-prior=1, slow=0
  Delete               : 0.05, jacobian=1,     alters-prior=1, slow=0
  Merge                : 0.05, jacobian=0.125, alters-prior=1, slow=0
  Split                : 0.05, jacobian=8,     alters-prior=1, slow=0
  Alter Radius         : 0.3,  jacobian=1,     alters-prior=1, slow=0
  Alter Position       : 0.4,  jacobian=1,     alters-prior=1, slow=0
  Slow Replace Position : 0.1,  jacobian=1,     alters-prior=1, slow=1
======================================================================
Settings
======================================================================
  Prior Exponent      : 1
  Likelihood Exponent : 4.5
  Posterior Exponent  : 20

  Radius Mean                 : 12
  Radius std. dev.            : 2
  Mean num. features          : 55
  Circle Overlap Density      : 10
  Num. Sample Points          : 32
  Alter Position Std. Dev.    : 3
  Alter Radius Std. Dev.      : 2
  Split Move Radius Proportion  : 0.08
======================================================================
Performance Settings
======================================================================
  Speculative moves disabled (6 threads available)
  Speculative chains disabled

======================================================================

Image          : large-white-cells.jpg
```

```
Initial Model : <random> (55 features)
Target Model  : unavailable
Iterations    : 100,000

        Move Name Proposed Accepted Rejected Invalid % Accepted  Av. Time
----------------- -------- -------- -------- ------- ----------  --------
Add                   4955      116     4839       0     2.341% 1.657e-05
Delete                4946       34     4912       0    0.6874% 1.339e-05
Merge                 5089      121     4968       0     2.378% 1.889e-05
Split                 5003       17     4954      32     0.342% 2.301e-05
Alter Radius         30381     4649    25732       0      15.3% 1.719e-05
Alter Position       39874     4693    35069     112      11.8% 1.713e-05
Slow Replace Pos.     9932       36     9860      36    0.3638%  0.001299

Total               100180     9666    90334     180     9.666%

Prior term dominated in 22,902 / 100,000 move proposals (22.9%)
Total number of steps                  : 100000
Total number of iterations             : 100000
Average number of iterations per step  : 1 (sd=0)
Move rejection probability             : 0.90334
Average time per step (secs)           : 0.000144976 (sd=0.000392609)
Minimum step time (secs)               : 6e-06
Maximum step time (secs)               : 0.00311

33 features found in 15 seconds
Elapsed real-time: 14.5475
```

Bibliography

[1] Tinku Acharya and Ajoy K. Ray. *Image Processing, Principlies and Applications*. John Wiley & Sons, 2005.

[2] Fahimah Al-Awadhi, Christopher Jennison, and Merrilee Hurn. Statistical image analysis for a confocal microscopy two-dimensional section of cartilage growth. *Journal Of The Royal Statistical Society Series C*, 53:31–49, 2004.

[3] Gautam Altekar, Sandhya Dwarkadas, John P. Huelsenbeck, and Fredrik Ronquist. Parallel Metropolis-Coupled Markov chain Monte Carlo for Bayesian Phylogenetic Inference. Technical Report 784, Department of Computer Science, University of Rochester, July 2002.

[4] David P. Anderson. Boinc: A system for public-resource computing and storage. In *5th IEEE/ACM International Workshop on Grid Computing*, November 2004.

[5] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[6] Victor Ayala-Ramirez, Carlos H. Garcia-Capulin, Arturo Perez-Garcia, and Raul E. Sanchez-Yanez. Circle detection on images using genetic algorithms. *Pattern Recogn. Lett.*, 27(6):652–657, 2006.

[7] Massimiliano Bonamente, Marshall K. Joy, John E. Carlstrom, Erik D. Reese, and Samuel J. LaRoque. Markov chain Monte Carlo joint analysis of Chandra X-ray imaging spectroscopy and sunyaevzel'dovich effect data. *The Astrophysical Journal*, 614(1):56–63, 2004.

[8] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[9] J. M. R. Byrd, S. A. Jarvis, and A. H. Bhalerao. Reducing the run-time of MCMC programs by multithreading on SMP architectures. In *22nd IEEE International Symposium on Parallel and Distributed Systems (IPDPS)*, 2008.

[10] J. M. R. Byrd, S. A. Jarvis, and A. H. Bhalerao. Speculative moves: Multithreading Markov Chain Monte Carlo programs. In *High-Performance Medical Image Computing and Computer Aided Intervention (HP-MICCAI)*, 2008.

[11] J. M. R. Byrd, S. A. Jarvis, and A. H. Bhalerao. On the parallelisation of MCMC-based image processing. In *24th IEEE International Symposium on Parallel and Distributed Systems (IPDPS)*, 2010.

[12] J. M. R. Byrd, S. A. Jarvis, and A. H. Bhalerao. On the parallelisation of MCMC by speculative chain execution. In *24th IEEE International Symposium on Parallel and Distributed Systems (IPDPS)*, 2010.

[13] Richard H. Carver and Kuo-Chung Tai. *Modern multithreading: implementing, testing, and debugging multithreaded Java and C++/Pthreads/Win32 Programs.* John Wiley & Sons, 2006.

[14] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming.* The MIT Press, October 2007.

[15] Mary Kathryn Cowles and Bradley P. Carlin. Markov chain Monte Carlo convergence diagnostics: A comparative review. *Journal of the American Statistical Association*, 91(434):883–904, 1996.

[16] Circle detection using Fast Finding and Fitting (FFF) algorithm. Circle detection using fast finding and fitting (fff) algorithm. *Geo-Spatial Information Science*, 3(1):74–78, March 2000.

[17] Jurgen A Doornik, Neil Shephard, and David F Hendry. Parallel computation in econometrics: A simplified approach. Open access publications from university of oxford, University of Oxford, 2006.

[18] Alexei Drummond and Andrew Rambaut. Beast: Bayesian evolutionary analysis by sampling trees. *BMC Evolutionary Biology*, 7(1):214, 2007.

[19] Ian Dryden, Rahman Farnoosh, and Charles Taylor. Image segmentation using Voronoi polygons and MCMC, with application to muscle fibre images. *Journal of Applied Statistics*, 33(6), 2006.

[20] Ayres C. Fan, John W. Fisher, William M. Wells, James J. Levitt, and Alan S. Willsky. MCMC curve sampling for image segmentation. In *MICCAI 2007*, 2007.

[21] Denis Che Keung Fan. *Bayesian inference of vascular structure from retinal images*. PhD thesis, University of Warwick, May 2006.

[22] Xizhou Feng, Duncan A. Buell, John R. Rose, and Peter J. Waddell. Parallel algorithms for bayesian phylogenetic inference. *J. Parallel Distrib. Comput.*, 63(7-8):707–718, 2003.

[23] Michael J. Flynn. *Computer architecture: pipelined and parallel processor design*. Jones and Bartlett, 1995.

[24] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 2.1.* High Performance Computing Center Stuttgart (HLRS), June 2008.

[25] I. Foster. Globus toolkit version 4: Software for service-oriented systems. In *IFIP International Conference on Network and Parallel Computing*, pages 2–13. Springer-Verlag, 2006.

[26] Guang R. Gao, Mitsuhisa Sato, and Eduard Ayguadé, editors. *The International Journal of Parallel Programming*, volume 36. Springer Netherlands, June 2008.

[27] C. J. Geyer. Markov chain monte carlo maximum likelihood. *Interface Proceedings*, 1991.

[28] C. J. Geyer and E. A. Thompson. Annealing markov chain monte carlo with applications to ancestral inference. *Journal of the American Statistical Association*, 90(431):909–920, 1995.

[29] Walter R. Gilks, Sylvia Richardson, and D. J. Spiegelhalter. *Markov chain Monte Carlo in practice.* Chapman and Hall, 1996.

[30] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. Open MPI: A flexible high performance MPI. In *Proceedings, 6th Annual International Conference on Parallel Processing and Applied Mathematics*, Poznan, Poland, September 2005.

[31] Peter J. Green. *Practical Markov Chain Monte Carlo.* Chapman and Hall, 1994.

[32] Peter J. Green. Reversible jump Markov Chain Monte Carlo computation and Bayesian model determination. *Biometrika*, 82:711–732, 1995.

[33] Peter J. Green. MCMC in action: a tutorial. given at ISI, Helsinki, August 1999.

[34] Peter J. Green and Antonietta Mira. Delayed rejection in reversible jump metropolis-hastings. *Biometrika*, 88(4):1035–1053, December 2001.

[35] Geoffrey Grimmett and David Stirzaker. *Probability and random processes.* Oxford University Press, 2001.

[36] M. Harkness and P. Green. Parallel chains, delayed rejection and reversible jump MCMC for object recognition. In *British Machine Vision Conference*, 2000.

[37] W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[38] John P. Huelsenbeck and Fredrik Ronquist. MRBAYES: Bayesian inference of phylogenetic trees. *Bioinformatics*, 17(8):754–755, 2001.

[39] John P Huelsenbeck and Fredrik Ronquist. MrBayes: A program for the Bayesian inference of phylogeny. Technical report, Department of Biology, University of Rochester, 2003.

[40] Bernd Jähne. *Digital image processing.* Springer, 6 edition, 2005.

[41] Michael Johannes and Nicholas Polson. Mcmc methods for financial econometrics. In *Handbook of Financial Econometrics.* North-Holland. Forthcoming, 2002.

[42] W S Kendall, F Liang, and J-S Wang. *Markov Chain Monte Carlo: Innovations and Applications.* World Scientific Publishing Co., 2005.

[43] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[44] Shuying Li, Dennis K. Pearl, and Hani Doss. Phylogenetic tree construction using Markov chain Monte Carlo. *Journal of the American Statistical Association*, 1999.

[45] Colin C. McCulloch. *High Level Image Understanding via Bayesian Hierarchical Models*. PhD thesis, Institute of Statistics and Decision Sciences, Duke University, 1998.

[46] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[47] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge University Press, 1995.

[48] Girija J. Narlikar and Guy E. Blelloch. Pthreads for dynamic and irregular parallelism. In *In Proc. of Supercomputing 98*, pages 4–1. IEEE, 1998.

[49] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads programming*. O'Reilly & Associates, 1996.

[50] David A. Patterson and John L. Hennessy. *Computer organization and design: the hardware/software interface*. Morgan Kaufmann, 2009.

[51] G Perrin, X Descombes, and J Zerubia. A marked point process model for tree crown extraction in plantations. In *IEEE International Conference on Image Processing*, volume 1, pages 661–4, 2005.

[52] Christian P. Robert and George Casella. *Monte Carlo statistical methods*. Springer, 2004.

[53] Jeffrey S. Rosenthal. Parallel computing and Monte Carlo algorithms. *Far East Journal of Theoretical Statistics*, 4:207–236, 2000.

[54] Kevin Smith. *Bayesian methods for visual multi-object tracking with applications to human activity recognition.* PhD thesis, Ecole Polytechnique Fdrale de Lausanne, Lausanne, 2007.

[55] Jeffrey M. Squyres and Andrew Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy, September / October 2003. Springer-Verlag.

[56] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Condor – a distributed job scheduler. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux.* MIT Press, October 2001.

[57] E. Thonnes, A. Bhalearo, W. S. Kendall, and R. Wilson. Bayesian analysis of vascular structure. In R. G. Ackroyd, K. V. Mardia, and M. J. Langdon, editors, *Stochastic Geometry, Biological Structure and Images*, pages 115–118, 2003.

[58] E Thonnes, A. H. Bhalerao, W Kendall, and R Wilson. A Bayesian approach to inferring vascular tree structure from 2D imagery. In *International Conference on Image Processing*, volume 2, pages 937–940, 2002.

[59] High Throughput, Monte Carlo, Todd Tannenbaum, Jim Basney, Rajesh Raman, and Miron Livny. High throughput monte carlo, 1999.

[60] L. Tierney and A. Mira. Some adaptive Monte Carlo methods for Bayesian inference. *Statistics in Medicine*, 18:2507–2515, 1999.

[61] David A. van Dyk and Hosung Kang. Highly structured models for spectral analysis in high-energy astrophysics. *Statistical Science*, 19(2):275–293, 2004.

[62] George Vogiatzis, Philip Torr, and Roberto Cipolla. Bayesian stochastic mesh optimisation for 3D reconstruction. In Richard Harvey and J. Andrew Bang-

ham, editors, *British Machine Vision Conference*, volume 2, pages 709–718, 2003.

[63] Eric W. Weisstein. Circle-circle intersection. From MathWorld–A Wolfram Web Resource, 4 2010.

[64] Eric W. Weisstein. Normal distribution. From MathWorld–A Wolfram Web Resource, 4 2010.

[65] Eric W. Weisstein. Poisson distribution. From MathWorld–A Wolfram Web Resource, 4 2010.

[66] Dan Werthimer, Jeff Cobb, Matt Lebofsky, David Anderson, and Eric Korpela. Seti@home—massively distributed computing for seti. *Computing in Science and Engg.*, 3(1):78–83, 2001.

[67] Darren J. Wilkinson. Bayesian methods in bioinformatics and computational systems biology. *Brief Bioinform*, 8(2):109–116, 2007.

[68] HK Yuen, J Princen, J Illingworth, and J Kittler. Comparative study of hough transform methods for circle finding. *Image and Vision Computing*, 8(1):71 – 77, 1990.

[69] Tao Zhao and R. Nevatia. Tracking multiple humans in crowded environment. In *Computer Vision and Pattern Recognition*, volume 2, pages II–406–II–413 Vol.2, 2004.

[70] Tao Zhao and Ram Nevatia. Stochastic human segmentation from a static camera. In *MOTION '02: Proceedings of the Workshop on Motion and Video Computing*, page 9, Washington, DC, USA, 2002. IEEE Computer Society.