Western Graduate&PostdoctoralStudies

Western University
Scholarship@Western

Electronic Thesis and Dissertation Repository

9-12-2013 12:00 AM

Stochastic simulation and spatial statistics of large datasets using parallel computing

Jonathan SW Lee
The University of Western Ontario

Supervisor
Dr. Reg Kulperger
The University of Western Ontario Joint Supervisor
Dr. Hao Yu
The University of Western Ontario

Graduate Program in Statistics and Actuarial Sciences
A thesis submitted in partial fulfillment of the requirements for the degree in Doctor of Philosophy
© Jonathan SW Lee 2013

Follow this and additional works at: https://ir.lib.uwo.ca/etd

Part of the Applied Statistics Commons, and the Other Statistics and Probability Commons

Recommended Citation
Lee, Jonathan SW, "Stochastic simulation and spatial statistics of large datasets using parallel computing" (2013). *Electronic Thesis and Dissertation Repository*. 1652.
https://ir.lib.uwo.ca/etd/1652

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact wlswadmin@uwo.ca.

STOCHASTIC SIMULATION AND SPATIAL STATISTICS OF LARGE DATASETS USING PARALLEL COMPUTING

(Thesis format: Monograph)

by

Jonathan Lee

Graduate Program in Statistical and Actuarial Sciences

A thesis submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

© Jonathan S. W. Lee 2013

Abstract

Lattice models are a way of representing spatial locations in a grid where each cell is in a certain state and evolves according to transition rules and rates dependent on a surrounding neighbourhood. These models are capable of describing many phenomena such as the simulation and growth of a forest fire front. These spatial simulation models as well as spatial descriptive statistics such as Ripley's K-function have wide applicability in spatial statistics but in general do not scale well for large datasets. Parallel computing (high performance computing) is one solution that can provide limited scalability to these applications. This is done using the message passing interface (MPI) framework implemented in R through the `Rmpi` package. Other useful techniques in spatial statistics such as point pattern reconstruction and Markov Chain Monte Carlo (MCMC) methods are discussed from a parallel computing perspective as well. In particular, an improved point pattern reconstruction is given and implemented in parallel. Single chain MCMC methods are also examined and improved upon to give faster convergence using parallel computing. Optimizations, and complications that arise from parallelizing existing spatial statistics algorithms are discussed and methods are implemented in an accompanying R package, `parspatstat`.

Keywords: spatial statistics, lattice models, point processes, parallel computing, high performance computing, Markov Chain Monte Carlo

Acknowledgments

I would like to thank the faculty and staff in the Department of Statistical and Actuarial Sciences at the University of Western Ontario for making my years of graduate school feel warm and welcoming. In particular I would like to thank the following people:

My Ph.D. supervisors Dr. Hao Yu and Dr. Reg Kulperger for their guidance and inspiration. My M.Sc. supervisors Dr. John Braun and Dr. Douglas Woolford for encouragement and opportunities. The department administrative staff, Jennifer Dungavell, Jane Bai, and Lisa Hines for putting up with my incessant questions and special requests. Dr. Bethany White and the staff at the Teaching Support Centre for their mentorship in teaching and training.

The writing of this thesis was made more enjoyable by the friends I am lucky enough to have in my life, both in London and Toronto.

But most important of all, I would like to thank my family for their continued support and inspiration through many years of schooling. I couldn't have done this without you.

Thank you all.

Contents

Abstract **ii**

Acknowlegements **iii**

List of Figures **vii**

1 Introduction **1**

1.1 Spatial point processes 1
1.1.1 Homogenous Poisson process 4
1.1.2 Inhomogenous Poisson process 5
1.1.3 Stationarity and isotropy 6
1.1.4 Summary statistics and summary functions 7
1.1.5 Point process model fitting and assessment 19
1.2 Stochastic lattice models 20
1.2.1 Markov random fields 21
1.3 Parallel computing 22
1.3.1 Hardware specifications 25
1.3.2 Parallel programming in R 26
1.3.3 Embarrassingly Parallel and Non-embarrassingly Parallel Problems 27
1.4 Motivation for parallel computing in spatial statistics 30
1.4.1 Use of spatial statistics in different disciplines 30
1.4.2 Parallel computation in spatial statistics 31
1.5 Outline of Thesis 33

2 Issues in parallel computing **34**

2.1 Issues with Parallelizing Single Simulations 34
2.1.1 Message Passing Interface 35
2.1.2 Random number generation 35

2.1.3 Workload distribution 38
2.1.4 Example: Two-dimensional Heat Diffusion System 39
2.2 Issues in Parallel Computing of Spatial Statistics 41
2.2.1 Memory limitations 41
2.2.2 A virtual topography division 43
2.2.3 Complex spatial windows 45
2.3 Other issues in parallel computing 49
2.3.1 Multiple layers of workers 49
2.3.2 Optimal number of processors 50
2.3.3 Load balancing 51
2.4 A simulation model of optimal parallel computing structure 54
2.4.1 A simple model 54
2.4.2 A simple model with concurrent job distribution and load balancing 56

3 Parallel computing for spatial point processes 59
3.1 Edge correction methods 59
3.1.1 Comparison of edge correction methods 69
3.2 Parallel computation of point process summary functions 70
3.3 Performance benchmarks 75
3.4 Point process model fitting and evaluation 77
3.4.1 Simulation envelopes 79
3.5 Point process reconstruction 80
3.5.1 The general reconstruction algorithm 81
3.5.2 An improved reconstruction algorithm 87
3.5.3 Parallelizing reconstruction 89
3.6 The *parspatstat* package 94
3.6.1 Function usage 95
3.6.2 Datasets that do not fit in memory 97
3.7 Example: Ontario lightning data 98

4 Parallel computing for lattice models 109
4.1 Motivating example: Lattice Fire Spread Model 109
4.2 Benchmarking and scaling issues 110
4.3 Optimizations 114
4.3.1 Time barrier 114
4.3.2 Irregular division of lattice 116

4.3.3 Example: Interacting Particle System 116
4.3.4 Boundary buffer zones 117
4.4 Parallel computing for Markov Chain Monte Carlo 118
4.4.1 Multiple chain MCMC 121
4.4.2 Single chain MCMC 124
4.4.3 Continuous pre-fetching 129
4.4.4 Comparison with other parallelization techniques 131

5 Conclusion **132**
5.1 Further Work 133

Bibliography **135**

Curriculum Vitae **140**

List of Figures

1.1 Summary functions plotted for a realization of a homogenous Poisson process with intensity $\lambda = 100$ in a unit window. The top-left graphic is a plot of the realized poisson pattern. 14
1.2 When looking at a search radius that lies outside the boundary of the spatial window, no points exist in this censored region (solid red), leading to a biased estimate of summary functions. 17
1.3 A manager-worker model for parallel computing where a single processor (the manager) is dedicated to distributing work to and consolidating results from s processors (the workers). Communication can occur between manager and workers but also between workers themselves in some circumstances. 24
1.4 Parallelizing multiple simulations by distributing independent simulations to multiple CPUs. 29
1.5 Parallelizing a single simulation by distributing smaller dependent parts to multiple CPUs. 29

2.1 Division of lattice from 1 to 9 processors, maintaining uniform sub-lattices. 46
2.2 Division of lattice from 1 to 9 processors, minimizing sub-lattice boundaries. 47
2.3 Benchmark of the number of jobs vs computation time on two different point patterns. 53
2.4 Benchmark of the number of jobs (in excess of the number of workers) vs time with load balancing (red line) and without load balancing (black line) for varying data sizes. 58

3.1 Toroidal edge correction on a point pattern (solid points) by replicating the point pattern 8 times (hollow points) but only treating the original points as centers. 61

3.2 Border method edge correction on a point pattern by discarding points within some maximum search radius from an edge (hollow points). Only interior points (solid points) are used as centers. In this example, it resulted in 74% of points being discarded using a maximum search radius of 1/4 of the square window dimensions. 64
3.3 The border method may not work well for even a simple window shape if it needs to be reduced by the some maximum search radius (left). It may result in disjoint interior windows if the maximum search radius is too large (right). 65
3.4 The plot of *cells* data (left) and its corresponding Fry plot (right). . . 67
3.5 Smoothed estimates of the reduced second moment measure, from which an estimate of the K function can be obtained at varying radius distances from the origin. 67
3.6 Estimated variances from simulation of binomial processes with varying n under border, isotropic, and translation correction methods. 71
3.7 A simple example of dividing a point pattern (above) into two sub-windows (below). Although the spatial window is divided by half, points that are within the area covered by the maximum search radius (area enclosed by dotted red line) still need to be stored. 73
3.8 When applying edge correction in parallel computation of spatial statistics, each processor is aware of only the extent of its own sub-window, yet needs to be aware of whether a boundary is a global boundary (black) and requires edge correction, or an interior boundary (red) that requires no edge correction. 74
3.9 Benchmark of the number of CPU vs time for uniform point process patterns of varying size. 76
3.10 Speed up of parallel K-estimate for point patterns of varying sizes. . 77
3.11 The observed point pattern and a reconstructed point pattern (top). Energy function of the reconstructed point pattern over 1,500 iterations (bottom). 85
3.12 The observed point pattern (left) and a balanced reconstructed point pattern (right). 89
3.13 Comparison of simulation envelopes of K-function for the observed point pattern, reconstructed point pattern, and a balanced reconstructed point pattern. Bottom graph shows the effect of a balanced algorithm in better matching at small values of r. The top and bottom lines of each colour indicate the upper and lower bounds of each envelope. . . 90

3.14 Energy function decrease for the reconstruction algorithm parallelized on various number of processors. 93
3.15 Chunking in a dataset and distribution to workers. 99
3.16 Map of all lightning strike data and the extent of the extracted spatial window. The extracted spatial window is plotted with points intead of a + symbol and thus appears lighter. 100
3.17 Summary of Ontario lightning strike data from 1992 to 2010 by year (top) and by month (bottom). 101
3.18 Plot of lightning strikes in a region of northern Ontario between 47° and 51° latitude and $-84°$ and $-80°$ longitude in 2003 (left) and 2008 (right). 103
3.19 Plot of K-estimates of lightning strikes in a square region of Ontario in 2003 (top) and 2008 (bottom) using various border correction methods along with the theoretical line. 104
3.20 The original and reconstructed point patterns (top) and the corresponding K-functions (bottom). 107
3.21 The original (left) and reconstruct point patterns (right) conditioning on the points in red. 108

4.1 Simulation of a simple fire spread model with uniform wind on a 120x120 lattice. Red dotted lines represent division of sub-lattices to 4 CPUs. 111
4.2 Computation time of 48 fires distributed amongst 2 to 24 CPUs on a 1,200 by 1,200 pixel lattice. 113
4.3 Computation time of 48 fires distributed amongst 2 to 24 CPUs on a 240 by 240 pixel lattice. 113
4.4 Illustration of 3 CPUs working in parallel with communication only when all three CPUs have hit a time barrier or requires swapping. . . 115
4.5 Illustration of 3 CPUs working in parallel with communication at every time step to synchronize. 115
4.6 Buffer zone to alleviate communication due to boundary crossing, but covers less total area. 119
4.7 Movement of a point (striped) from one processor to another location (solid). The second processor (blue) does not actually receive the point until it passes the buffer zone. 120

4.8 Metropolis trees with steps computed serially. At each time step, a single level is completed, representing one draw in the chain. Four draws are computed in four time steps. 123
4.9 Metropolis trees with prefetching two steps at a time using 3 processors. At each time step, two levels are completed, representing two draws in the chain. Four draws are completed in two time steps. 125
4.10 Metropolis trees with dynamic prefetching where the most likely paths are evaluated in parallel using 3 processors. At each time step, up to 3 levels are completed. In this particular example, five draws are completed in two time steps. 127
4.11 Comparison of speed-up versus serial MCMC from using up to 100 processors using simple pre-fetching and dynamic pre-fetching and two different targeted acceptance rates. 128
4.12 Metropolis trees with continuous prefetching where processors can be assigned to higher levels paths to more quickly determine which path is correct and then processors can be reassigned down further down the most probable paths. Asterisks indicate cells that are being computed in parallel and purple cells indicate cells that are already under evaluation. 130

5.1 Splitting a three dimensional lattice into smaller sub-lattices. 134

Chapter 1

Introduction

1.1 Spatial point processes

Spatial point processes form a large component of the field of spatial statistics. A spatial point process, N, is a model that aims to describe an observed point pattern. A realization or sample from N is a point pattern consisting of n points, each of which has a spatial location in d dimensions attached to it.

$$x_1, \ldots, x_n \in \mathbb{R}^d$$

In the remainder of this thesis, Cartesian coordinates in $d = 2$ dimensions (the planar case) are primarily used as the location measure, though other measures may be used (e.g. polar coordinates). In practice, d is rarely greater than 3 since point processes are usually used to describe a physical problem in up to three spatial dimensions.

A point pattern is observed within a spatial window, denoted E, which is a subset of the space on which the process is defined, $\mathbb{R}^d$. Often, the spatial window is rectangular or circular in shape for the sake of simplicity but can technically be any polygon. Complex polygon spatial windows arise in practice both naturally (geographic boundaries such as tree lines) and artificially (man made boundaries such as roads or political divisions). There are situations where a spatial window can exhibit an interior 'hole' or even an island within a hole. The number of points that exist in an arbitrary window, W, is denoted by the function $N(W)$ while the volume (area) of the same window is denoted by the function $A(W)$. Thus, we have $N(E) = n$ though in practice, the number of points in a window is usually not fixed beforehand.

In addition to a location, each point in a spatial point pattern can also have one or more measures attached, known as marks. These measures, the location themselves, and correlation structure between points serve to characterize a point pattern through summary statistics and functions, described in more detail in section 1.1.4.

When modelling point processes, we will make several assumptions. The first is

that it is assumed that points exist in continuous space, though depending on the sampling strategy of the data, these points may be discretized to a lattice structure. Lattice structures need to be handled differently and will be discussed in more detail in section 1.2. However, it should be noted that discretized data can be reduced to a marked point process using smoothing techniques, as has been done in image pixel analysis.

The second assumption is that we are only dealing with infinite point processes. That is, it is assumed that the underlying model generating the point pattern extends infinitely but we are only observing the points within a chosen spatial window, the choice of which is made such that the point pattern is homogeneous within the window or additional covariates are included to explain any heterogeneity that may exist.

A finite point process on the other hand, exists only within the spatial window with the distribution of points within the window perhaps dependent on proximity to the edge of the window. This is not to be confused with edge effects and edge correction methods that are discussed in section 3.1 on infinite point processes where points exist but are not observed outside the window.

1.1.1 Homogenous Poisson process

Before discussing the homogenous Poisson process, we will discuss the finite binomial process. The finite binomial process is the simplest point process where n points are distributed completely randomly in a given spatial window, E. The probability of a point x appearing in any interior window W is uniformly distributed with probability equal to the proportion of the area of W to the area of E:

$$P(x \in W) = \frac{A(W)}{A(E)} \text{ where } W \subset E.$$

When generalized to n points $x_1, \ldots, x_n$, these points are independent and uniformly distributed in E when for all interior windows $W_1, \ldots, W_n$,

$$P(x_1 \in W_1, \ldots, x_n \in W_n) = P(x_1 \in W_1) \ldots P(x_n \in W_n) = \frac{A(W_1) \ldots A(W_n)}{A(E)^n}.$$

To further generalize the binomial point process, we do not have to fix the number of points in the point process, n. This leads to the homogeneous Poisson process if the number of points is Poisson distributed with some intensity parameter λ per unit area. Such a process is known as a completely spatially random (CSR) process where there is no clustering (points with a tendency to appear near each other) or regularity

(points appearing far from each other, also known as inhibition). The homogenous Poisson process is often the "benchmark" that various simulation and assessment methods use as a baseline comparison.

It should also be noted that the combination of many point patterns converges in behaviour to a Poisson process by the Poisson convergence theorem [21]. Also, a linear transformation of a Poisson process is still a Poisson process. In particular, if N is a homogenous Poisson process with intensity λ and B is a linear mapping, then $BN = \{Bx : x \in N\}$ is a homogenous Poisson process with intensity $\lambda \times det(B^{-1})$ [20].

1.1.2 Inhomogenous Poisson process

A homogenous Poisson process can be generalized to an inhomogenous Poisson process where the constant intensity λ is replaced by the intensity function $\lambda(x)$. The number of points in a window B is Poisson distributed with mean $\int_B \lambda(x)dx$ and the number of points in disjoint windows are independent. The second property is known as independent scattering. From a given point pattern, one can estimate the intensity function using parametric methods such as maximum likelihood estimation.

To generate from an inhomogenous Poisson process with intensity function $\lambda(x)$, one can first generate from a homogenous Poisson process with intensity equal to the maximum intensity of the desired intensity function. Then a rejection method can be

used to determine which points should be deleted (thinned) until the desired inhomogenous intensity surface is met. This is done by rejecting a point with probability equal to the proportion of its intensity to the overall maximum density.

The finite Cox process is a generalization of inhomogenous Poisson process where the intensity surface $\lambda(x)$ is random. It is a two stage process where a random intensity surface is generated, and then an inhomogenous Poisson process is constructed conditional on the generated intensity surface. This is also known as a doubly stochastic Poisson process.

To determine if the intensity surface should be a random variable (that is, to justify the use of a Cox process), multiple samples in the same window are necessary. The intensity function can be estimated for each sample and compared.

1.1.3 Stationarity and isotropy

The concepts of stationarity and isotropy are important when dealing with point processes since they are often assumed to be true when using the methods described in this thesis.

A stationary point process, also known as an homogenous process, is a point process, N, where the distribution of the number of points in a given window is the

same regardless of how that window is translated. That is,

$$N + \alpha \stackrel{d}{=} N \text{ for all } \alpha \in \mathbb{R}^d. \tag{1.1}$$

where if $N = \{x_1, x_2, \ldots\}$ then $N + \alpha = \{x_1 + \alpha, x_2 + \alpha, \ldots\}$.

A related concept to stationarity is isotropy. A point process is isotropic if it the distribution of the number of points in a given window is the same regardless of how that window is rotated. That is,

$$N \stackrel{d}{=} R_\beta N \text{ for all } \beta \text{ where } R_\beta \text{ is a rotation of angle } \beta.$$

The aforementioned homogenous Poisson process is both stationary and isotropic. In other words, one would expect all sub-windows $W \subset E$ that are the same shape to exhibit the same spatial characteristics regardless of orientation and position in E whereas an inhomogenous Poisson process would by definition be non-stationary.

1.1.4 Summary statistics and summary functions

Superficial characteristics of a point pattern such as clustering or regularity can often be observed from visual inspection. These characteristics, as well as other less obvious

characteristics, can be measured through summary statistics and summary functions.

The most basic summary statistic is point intensity, defined as the expected number of points in a unit area of the point pattern. This is denoted as λ with corresponding estimator $\hat{\lambda}$, satisfying the following definition for the expected number of points in an arbitrary spatial window W:

$$E(N(W)) = \lambda A(W)$$

This intensity is often not known and the standard estimator is simply the number of points in the observation window per unit area,

$$\hat{\lambda} = \frac{N(E)}{A(E)}.$$

If counting the number of points in E is not possible since it may be too time consuming, other estimators of $\hat{\lambda}$ exist such as the distance method [11] and the point-quarter method [20]. Such methods also lend themselves to reconstruction applications, discussed in section 3.5. In the non-stationary case, the intensity is instead replaced by an intensity surface, $\lambda(x)$ with estimation often done using maximum likelihood and verified with bootstrap methods [20].

Going beyond single numeric summary statistics are summary functions (often functions of a search radius, r). Those summary functions that are all based on a count of the number of points around a "typical point" can be referred to as Palm characteristics. We use the notation

$$n_x(r) = N(b(x,r)\backslash\{x\}) \tag{1.2}$$

to denote the number of points in the point process N that is within a circle (we are concerned with $d = 2$ dimensions) of radius r centered at point x excluding point x itself. Assuming stationarity, by equation 1.1, we can translate all points to the origin, o, and study the characteristics of just $n_o(r)$ in the resulting point pattern to obtain measures for the original point pattern. Further examination of this is given in section 3.1 where edge correction methods are discussed.

In a spatial point pattern, several descriptive statistics of this nature can be used to describe the point process characteristics. These include first-order statistics such as the empty space function $F(r)$, the nearest-neighbor distance distribution function, $G(r)$, and second-order statistics such as Ripley's K-function [2] $K(r)$, and the pair correlation function, $g(r)$. These, in addition to the combination of the F- and G-function into what is called the J-function [2] $J(r)$ are described below.

For a given point pattern, these functions can be compared to the corresponding theoretical function for the homogeneous Poisson process as an indicator of inhibition, clustering, or lack thereof (complete spatial randomness). Their theoretical values for the homogeneous Poisson process are also given.

First-order summary functions

The F-function, known as the empty space function, denotes the probability that an area of radius r around a typical point contains at least one point, defined as

$$F(r) = 1 - P(n_o(r) = 0) \text{ for } r \geq 0.$$

The G-function, known as the nearest neighbour distance distribution function is the distribution of distances of a typical point to its nearest neighbour, not including the point itself, defined as

$$G(r) = P(n_o(r) > 0) \text{ for } r \geq 0.$$

Again, the origin is used assuming stationarity is satisfied. For the homogenous Poisson process, one would expect $F(r) = G(r)$. If the probability of having a point around an arbitrary point is smaller than the typical distance of a nearest neighbour, that is, $F(r) \leq G(r)$, then this would be indicative of regularity (inhibition). Likewise,

$F(r) \geq G(r)$ would be indicative of clustering. Lieshout and Baddeley defines a measure, J, based on these two functions [47], defined as

$$J(r) = \frac{1 - G(r)}{1 - F(r)} \text{ for } r \geq 0 \text{ and } F(r) \leq 1.$$

For this function, one would expect $J(r) = 1$ for a homogenous Poisson process, $J(r) \geq 1$ to be indicative of regularity, and $J(r) \leq 1$ to be indicative of clustering. It is important to note that a non-Poisson process can also result in $J(r) = 1$, that is, $J(r) = 1$ is a necessary but not sufficient condition for showing a process is a homogenous Poisson process [4].

Second-order summary functions

The first-order summary functions described above do not look beyond the nearest neighbour, potentially ignoring a lot of information. Second-order statistics are based on pairwise interpoint distances. Ripley's K-function [32] is a second-order descriptive statistic that is commonly used to measure homogeneity of spatial point patterns. That is, to determine if a point pattern with n points that lies within a spatial window $E \subset \mathbb{R}^2$ follows a spatially random process or if it is the result of a clustering or regular process. Tests of homogeneity are further discussed in section 1.1.5. Under

the assumptions of stationarity and isotropy, the function, $\lambda K(r)$, is the expected number of points within a distance r of a typical point.

$$\lambda K(r) = E(n_o(r)) \text{ for } r \geq 0.$$

and by dividing both sides by the intensity λ, we obtain a definition of the K-function

$$K(r) = \frac{E(n_o(r))}{\lambda} \text{ for } r \geq 0.$$

In a homogenous Poisson process, this is simply the area of the search circle, $K(r) = \pi r^2$. More points than expected, $K(r) \geq \pi r^2$, is indicative of clustering and less points than expected, $K(r) \leq \pi r^2$, is indicative of regularity.

Related to the K-function is the L-function introduced by Besag [6] which is a variance stabilized version of the K-function, defined as

$$L(r) = \left(\frac{K(r)}{\pi} \right)^{1/2} \text{ for } r \geq 0.$$

which has an added benefit of being easier to interpret because $K(r) = \pi r^2$ is the area of the search circle with radius r so the L-function for a completely spatially random process is simply, $L(r) = r$. Hence, $\widehat{L(r)} - r$ is often plotted against $L(r) - r = 0$ to

visually examine the nature of the point process. Deviations from a horizontal line are easier to spot than deviations from a diagonal [20].

Figure 1.1 shows the aforementioned summary functions plotted for a realization of a homogenous Poisson process with intensity $\lambda = 100$ in a unit window compared with the corresponding theoretical summary functions for a homogenous Poisson process.

The pair correlation function, $g(r)$, is another second-order summary function that contains the same information as the K- and L- functions but has advantages for graphical interpretation. It is defined as

$$g(r) = \frac{k(r)}{2\pi r} \text{ for } r \geq 0$$

where $k(r)$ is the derivative of $K(r)$. That is, it satisfies

$$K(x) = \int_{-\infty}^{x} k(t)dt.$$

The pair correlation function is the correction factor of a point x in $b(x)$ (with probability λdx) and a point y in $b(y)$ (with probability λdy) where x, y are distance r apart and $b(x)$ is the infinitesimally small sphere (or disc) around a point x. The

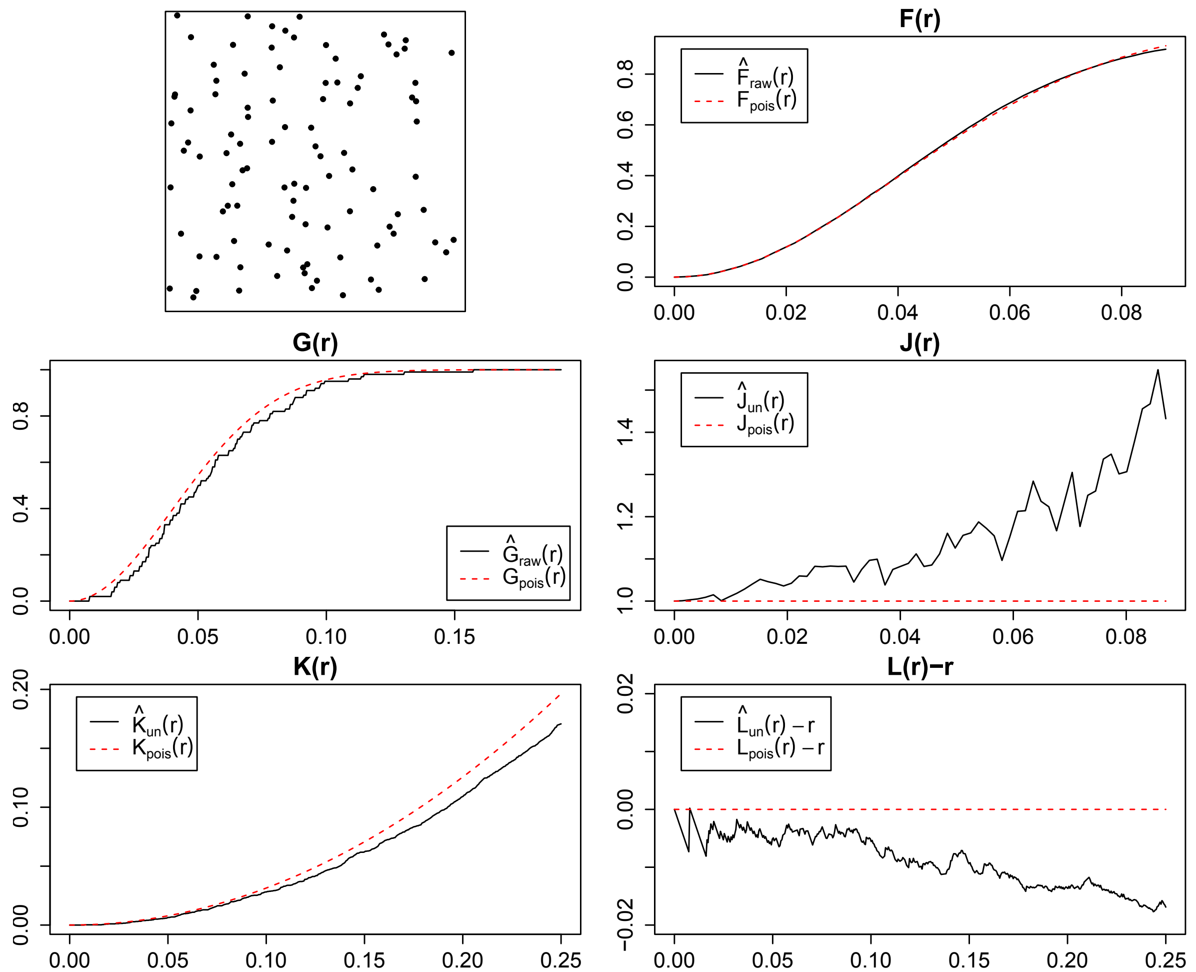

Figure 1.1: Summary functions plotted for a realization of a homogenous Poisson process with intensity $\lambda = 100$ in a unit window. The top-left graphic is a plot of the realized poisson pattern.

probability of both x in $b(x)$ and y in $b(y)$ is thus,

$$p_2(r) = g(r) \cdot \lambda dx \cdot \lambda dy$$

so for large radius r, points are expected to be independent so $g(r)$ approaches 1. In a cluster process, $g(r) \geq 1$ since points x, y are correlated.

Higher order summary statistics have been developed as well, for example Schladitz and Baddeley's T function that looks at the number of r-close triples of points per unit area [36]. There are also topological summary characteristics that make use of methods from other areas such as graph theory. The connectivity function for example, $c(r)$, measures the number of disjoint components when circles of a selected radius R are grown around each point and the union of all circles are taken as components. The actual measure itself is taken as the probability of a point distance r from a typical point to be in the same component. Near $r = 0$, the number of components will be close to the number of points and this function approaches 0 monotonically for increasing r.

Other examples include detecting outliers in point patterns by statistically analyzing marks assigned to each point through point-based indices. Gaps in the data can be detected using network graphs created by joining k nearest neighbours and

then analyzing the areas of the resulting cells created by the graph edges. In this thesis, we will primarily focus on first- and second-order summary characteristics with a particular emphasis on Ripley's K-function as these are the ones that are most commonly used in practice.

Edge correction methods

When computing the summary functions defined above, they all involve looking at a circle of some radius r around a given point, namely $n_o(r)$. A complication arises when a point is within r distance from the boundary of the spatial window E so that part of the search circle lies outside of E where no points are observed (Figure 1.2). We will illustrate this point further in section 3.1 by examining the K-function where this censored data creates a biased estimate for which we need to use edge correction methods. In fact, the K-function is independent of the shape of the study area when edge effects are corrected for properly [10].

The uncorrected K-function estimate is known as the naive estimator where we have

$$\hat{K}(r) = \frac{1}{\lambda n} \sum_{i=1}^{n} n_i(r) \text{ for } r \geq 0.$$

If we include a 'buffer' area around our window of interest for which data is collected as well, this can describe the spatial pattern most accurately, but may not warrant

the additional labor involved [17]. Various edge correction methods are described and compared in section 3.1. Also of interest is the J-function, for which it was shown by Baddeley that estimates of the J-function even without applying edge correction are unbiased [2].

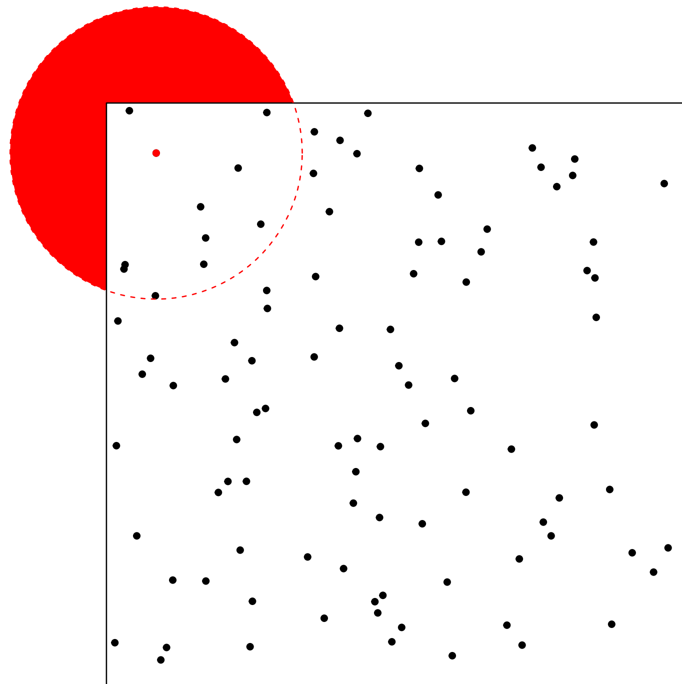

Figure 1.2: When looking at a search radius that lies outside the boundary of the spatial window, no points exist in this censored region (solid red), leading to a biased estimate of summary functions.

Other summary characteristics

In addition to the described nearest neighbour and intensity statistic, researchers have also defined indices that can be used as summary characteristics.

The index of dispersion is a location-based index that measures the ratio of the variance of the number of points to the mean number of points in enclosed spatial windows of a certain size and shape. Pielou's index of randomness is also based on location of points and is a function of the random distance from a test point to its nearest neighbour. These location-based indices extend well to marked point patterns because they can be computed independent of the mark.

Point related indices are those that assign a mark to each point based on some characteristic of the point in relation to the pattern as a whole. An example is the aggregation index which marks each point with its nearest neighbour distance and aggregates these marks over the entire set to get a measure of average nearest neighbour distance. Related to the aggregation index is the degree of colocalisation which is a function of some search radius r and gives the proportion of points that have an aggregation index mark smaller than r. The mean direction index marks each point by the sum of the unit vectors from the point to its k nearest neighbours. This gives the general direction that a point's nearest neighbours are in. If a point has nearest neighbours with opposing vectors, then this gives a direction index of 0.

1.1.5 Point process model fitting and assessment

Given a point pattern, a common question of interest is whether or not the pattern exhibits one or more characteristics of clustering, regularity, and randomness. Note that it is possible for a point pattern to have short range clustering yet long ranger regularity. To test this, we compute summary statistics of the point pattern and compare that with the same summary statistic computed either on realizations of a homogeneous Poisson process model or analytically on the theoretical homogenous Poisson process model. If the results are statistically different in a two-sided test, then it is evidence against complete spatial randomness. A one-sided test would indicate either clustering or regularity. This result is necessary, but not sufficient proof of complete spatial randomness and multiple tests are recommended. In practice, functional summaries like the K-function or its variance stabilized version, L-function, are used with the maximum point-wise difference with the same function computed on a realization of the point process as an approximation to the theoretical value. The advantage of comparing with realizations from a point process model is that it can easily be extended to more complicated point process models as well since theoretical values of many summary functions can only be analytically determined for simple point processes. Beyond comparing observed point pattern characteristics to a completely spatially random process, one can compare with other fitted point

process models, which will not be discussed in detail here.

The choice of maximum search radius distance is important as well since variance at larger radius will be high, thus weakening the power of the test. In order to account for this, Ho and Chiu [19] proposes a modified L-test with a weight function for higher search radii r. A comparison of the variance as a function of the choice of radius on different edge correction methods is given in section 3.1.1 to demonstrate the larger variance problem at larger radiuses.

A resampling method can be used as a means to assess the fit of a model as well. Simulated realizations from a proposed model are generated and summary functions are computed for each realization. Point-wise or global quantiles for these can be used to construct an envelope to observe if the summary function for the observed point pattern fits within this envelope. Further details for such methods are given in section 3.4.

1.2 Stochastic lattice models

Stochastic lattice models are a way to represent spatial locations in a grid structure where each cell (pixel) is in a state and evolves according to stochastic transition rules. These models can be used to describe many phenomena such as the growth

of a forest fire front [7]. They are also used in statistical physics for modelling ferromagnetism, in biology for predator-prey models, and in epidemiology for modelling disease transmission. The grid structure of the lattice model lends itself to take advantage of parallel computing. This parallelization allows us to decrease computation time which can allow us to increase resolution of the lattice, work with a larger lattice, decrease time step size in simulation, or work with a more complicated model.

Lattice models also have a direct translation to point processes models, notably, a reason for studying lattice-based processes is their relatively tractability by comparison with inhibition processes [11]. Assuming each lattice cell represents a point, then depending on how the position of a point is interpolated within a cell, this can naturally produce inhibition between points. For example, if we take the centre of each cell as the location of a point, then a hardcore inhibition distance equal to the cell dimensions will exist.

1.2.1 Markov random fields

Markov random field models were first introduced in 1974 by Besag [5] and were intended as a stochastic model to describe spatial processes of points represented in a lattice. A set of random variables are organized in a lattice structure so that any cell j of the lattice that is a neighbour of a cell i has the Markovian property

that the functional form of $P(x_i|x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$ is dependent upon x_j. In a square lattice structure, this creates a possible first-order neighbour structure of cells consisting of cells immediately to the top, left, right, and bottom of an existing cell. The value for a cell is dependent only on its neighbour cells. Higher order neighbour structures can be defined to extend beyond immediate neighbours as well but we will mainly look at the first-order neighbour structure as it is the most common.

Techniques discussed in Chapter 4 can be directly applied to a Markov random field existing on such a lattice structure if we wanted to simulate a stochastic realization. If we wish to analyze a Markov random field, doing so by direct calculation is very difficult. Even for an unrealistically small 40 x 40 pixel image where the pixels take on binary values, there are $2^{1600} = 4.4 \times 10^{481}$ terms in the summation [15]. Using a Bayesian approach, one can employ Markov Chain Monte Carlo (MCMC) techniques such as those described by Givens and Hoeting [15] to do so. A description of MCMC and techniques for parallelizing MCMC are given in section 4.4.

1.3 Parallel computing

For the past several years, computing power has plateaued due to physical constraints on the design of microchips which limits frequency scaling as a means of increasing

computing power. Instead, parallel computing, which uses multiple processors to work concurrently to speed computation, has been the focus of research and development in recent years. From a statistical computing perspective, high-performance computing (HPC) that makes use of computer clusters consisting of thousands of cores is one of the primary tools to compute intensive simulations and calculations.

Two architectures of parallel computing are multithreading and multiple processing. The main difference between multithreading and multiple processing is the memory architecture. Multithreading involves dividing a single process into the work of multiple computing threads. Each of these threads share the same memory architecture and usually exist on a single computer system. Multiple processing on the other hand divides a problem into smaller problems that are then worked on independently by separate processors. Each processor sees only its own memory structure and if required, can exchange information with other processors. Multithreading provides a simpler method of parallelizing problems due to all threads working on the same data (equal access to shared memory) but is limited in the amount of speed-up possible for memory intensive computations. Multiple processing can scale linearly (ignoring communication overhead) with the number of processors used but is more complicated to implement. Although communication overhead will always exist, it can be considered negligible relative to the intensive computation at hand. For the purposes of

this paper, multiple processing is the primary focus though there is room to use multithreading within the computation of individual processors such as parallel matrix computations implemented by libraries that can take advantage of multithreading.

In this thesis, we will be using a manager-worker model for parallel computing where a single manager processor is dedicated to distributing work to and synchronizing any number of worker processors (Figure 1.3). Although workers mostly communicate with the manager, it is possible for workers to directly communicate with one another to avoid having a communication bottleneck at the manager level. Most problems will have one layer of workers although technically each worker can in turn be a manager with its own set of workers. For example, we may wish to accommodate large datasets at each division of workers. Such a multi-layer division is described in section 2.3.1.

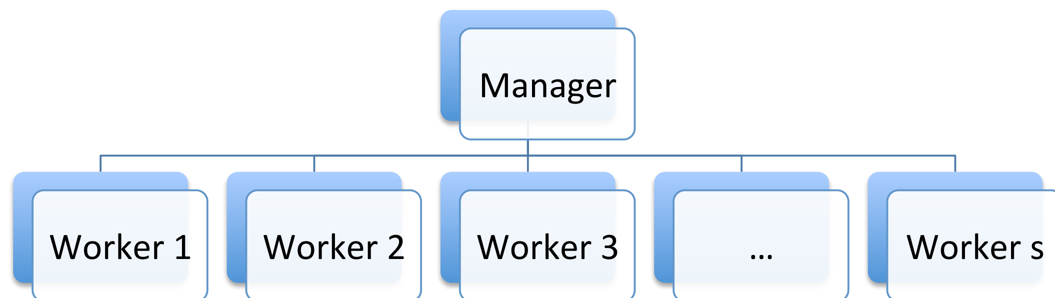

Figure 1.3: A manager-worker model for parallel computing where a single processor (the manager) is dedicated to distributing work to and consolidating results from s processors (the workers). Communication can occur between manager and workers but also between workers themselves in some circumstances.

1.3.1 Hardware specifications

In this thesis, simulations were carried out on the **mako** cluster on the shared hierarchical academic research computing network (SHARCNET). This particular cluster contains only 244 cores and is rarely used relative to larger clusters with thousands of cores. It is meant for development and testing purposes so each job has a one-hour limit on run time. Such an environment is not ideal for running actual analyses that may require longer run time but is suitable for performing benchmarking tests on a varying number of processors. However for this same reason, the “wall clock” time of jobs was kept to under an hour when run on a single processor so that the exact same computations can be run on multiple processors and their results can be fairly compared. In this thesis, the terms cores, processors, and CPUs are used interchangeably to denote the number of individual central processor units utilized in a parallel system.

The cluster runs on CentOS 5.2 and consists of four types of nodes: 1 administrative node, 1 login node, 14 Xeon computation nodes and 16 Nehalem computation nodes. Care was taken to carry out all benchmarks on only one type of node (Xeon nodes, in this case) to ensure uniformity between hardware. Nodes are interconnected through gigabit ethernet and jobs were kept to as few nodes as possible to minimize internode communication times. Each Xeon node consists of four Intel Xeon cores

operating at 3.0 GHz with 8GB of memory per node.

1.3.2 Parallel programming in R

R is an open source implementation of S, a programming language for statistical computing and graphics [31]. It is widely used both in practice and research in many fields across industry and academia. The core R software is supported by an active development team while external libraries that extend the functionality of R are developed and maintained by the community. As of writing, there are currently over 4,2800 of these external libraries, known as packages, on the Comprehensive R Archive Network (CRAN).

Support for parallel computing has been available in R for some time, Schmidberger et al. provides an excellent state of the art review [37]. Of note are two packages on which many of the existing parallel computing packages are built: `multicore` [44] and `Rmpi` [50]. To summarize, the `multicore` package allows for single computers to make use of all available cores on the machine, commonly two to eight cores, while `Rmpi` is designed to work on computing clusters with many cores spread out across many machines (nodes), though it is also capable of being run on a single computer with multiple cores. The widely used `snow` package [45] can extend the `Rmpi` package to ease in the setup and execution of parallel code.

Versions after R 2.14.0 include the `parallel` package [30] as one of the base packages in R. This package is a derivative of `multicore` and `snow` to further simply the end user's process of adapting their code to take advantage of parallel programming.

A more recent package, `pbdMPI` [26], focuses on "pretty big data" using the MPI framework. The idea here is to have each worker process perform computations on its own portion of a data set without the necessity of a managing computer controlling their behaviour. The advantage of this is allowing enormous data sets to be used since the data set does not need to be transferred to each worker individually. Individual results from workers can then be aggregated together to get the desired final result. However, this package assumes the problem is embarrassingly parallel, as described below.

1.3.3 Embarrassingly Parallel and Non-embarrassingly Parallel Problems

When applying parallel computing to statistics, problems can be divided into two groups: embarrassingly parallel problems and non-embarrassingly parallel problems.

Embarrassingly parallel problems consist of running multiple independent simulations or calculations that can be more-or-less blindly distributed amongst many

processes and then results are combined at the end (Figure 1.4). They are known as embarrassingly parallel problems due to how simple it is to conceptualize and in many cases, implement. In fact, all of the aforementioned parallel packages were developed for the embarrassingly parallel problems and are relatively straightforward to use.

In fields like spatial statistics, a problem may arise where computation on one location of our spatial dataset may be dependent on the concurrent computation of another location in our spatial dataset. This dependency requires workers to communicate with one another in an efficient way without creating long stalls in computation. These are referred to as non-embarrassingly parallel problems. An example of this is when we wish to apply parallel computing to a single large simulation of a lattice model. We can divide the lattice into smaller sub-lattices and then have each processor perform computations on a single sub-lattice (Figure 1.5) but we need to maintain communication between processes that are responsible for adjacent sub-lattices.

Specific issues that arise when parallelizing computation in spatial statistics are discussed in chapter 2.

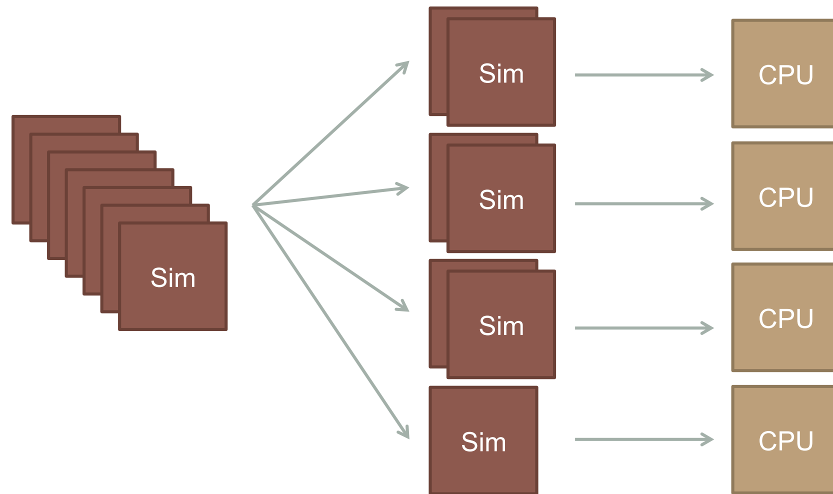

Figure 1.4: Parallelizing multiple simulations by distributing independent simulations to multiple CPUs.

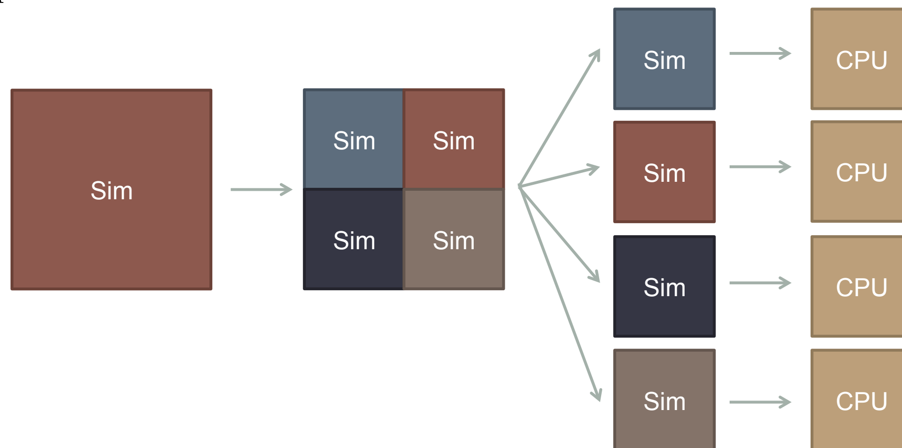

Figure 1.5: Parallelizing a single simulation by distributing smaller dependent parts to multiple CPUs.

1.4 Motivation for parallel computing in spatial statistics

This section serves to give an overview some of what has been done in parallel computing applied to various areas of spatial statistics as well as examples of applied statistical work that could potentially take advantage of the methods introduced in this thesis.

1.4.1 Use of spatial statistics in different disciplines

In ecology, Hasse [17] gives an overview of how to use the K-function in an ecological setting along with descriptions of edge correction methods. The methods were applied to both a real scrubland dataset and a computer generated dataset. Notably, he finds that the most effective edge correction methods are also the ones that have the highest computational requirement and points out shortcomings of the border method and toroidal correction. He also notes the non-standardized implementation of statistical analysis methods by other researchers that are not easily reproducible as many of the computer programs are either developed by the researchers themselves or modified programs from colleagues. One goal of this thesis is to provide an open-source library of some commonly used statistics implemented for parallel systems that can be easily

used and even built upon by anyone.

In forestry, Szwagrzyk et al. [43] examined the spatial distribution of trees in East-Central European forests using the K-function with edge correction done using the border method in a circular spatial window. This was compared with the theoretical homogenous poisson model to determine if clustering or regularity exists. In their analysis and other analyses of a similar nature with which they compared results, all were done on a small spatial scale which give a very manageable number of points to work with. The resulting methods from this thesis aims to alleviate compromises that need to be made with regards to size of datasets. Stoyan and Penttinen [41] gives an excellent summary of how spatial point process methods have been used in various aspects of forestry, including the calculation of summary characteristics, the reconstruction of point patterns, and stochastic models for marked point patterns.

1.4.2 Parallel computation in spatial statistics

A lot of the research that has been published on the use of parallel computing in spatial statistics has been from a geography standpoint. This is unsurprising as geography is a large area where spatial statistics has been used for many years. In 1990, Griffith [16] discussed limitations to spatial statistics at the time, which were mainly a lack of computer software to support statistical analysis and the growth of

geographic data sets outpacing the numerical computation capabilities at the time. He notes the importance of developing statistical algorithms with parallel computing in mind to take advantage of scalable computational resources.

In 1995, Armstrong and Marciano [1] looked at the parallel computation of a measure of spatial association introduced by Getis and Ord in 1992 [14]. Here, they examined the algorithm used for computing the measure to identify bottlenecks that are computed serially. They implemented portions of the algorithm in parallel and noted that the performance gain with parallel processing increase with the size of the data. Likewise, computation time increases at a slower rate in a parallel implementation compared to a serial implementation across datasets of equal size. This agrees with conclusions drawn throughout this thesis.

For lattice models, Cannataro et al. [9] developed a software environment for the parallel modelling of cellular automata models where cells (pixels) evolve according to transition rules that are dependent on its immediate neighbours. They examine the hardware required to run such a model in parallel, separating the system design into graphical and computational components. Load balancing from distributing work evenly across processes was also considered but this only applies when the model is dependent on first order neighbours. There are other strategies for optimizing parallel computation of such cellular (lattice) models that are discussed further in 4.3.

1.5 Outline of Thesis

The rest of this thesis is divided into a further four chapters. Chapter 2 will give an overview of both general issues and issues specific to parallelizing spatial statistics that will be encountered. Chapters 3 and 4 give methods and examples of applying parallel computing to spatial point processes and stochastic lattice models respectively. These chapters also look at performance benchmarks and introduce novel methods for further optimization. The package that accompanies the methods described is also introduced here along with examples on real data. Finally, Chapter 5 is the concluding chapter with discussion of further work and natural extensions to these methods.

Chapter 2

Issues in parallel computing

2.1 Issues with Parallelizing Single Simulations

As mentioned in section 1.3.3 regarding non-embarrassingly parallel problems, there are computational issues that arise when dealing with sub-problems that are dependent on results or information from adjacent sub-problems. These issues include the actual programming implementation as well as computational bottlenecks that need to be considered.

2.1.1 Message Passing Interface

Communication between processors (CPUs) in parallel programming is accomplished by message passing. Message passing is a programming paradigm used widely on parallel computers, especially Scalable Parallel Computers with distributed memory and on Networks of Workstations [39]. Message Passing Interface (MPI) has been the standard specification for message passing libraries across many computing platforms. It specifies a framework of functions that can be used for communication between processors regardless of hardware implementation. The `Rmpi` package is the R interface for the MPI framework [50], which serves as a wrapper for the underlying MPI implementation such as MPICH2, LAM/MPI, and OpenMPI. The `Rmpi` package is commonly required by various packages that wish to take advantage of parallel computing in R. In this thesis, this package is built upon and widely utilize on top of OpenMPI, though it will work with other MPI implementations as well.

2.1.2 Random number generation

In stochastic models, properly generated pseudorandom numbers are a necessity. Sequences of pseudorandom numbers are generated based on a seed that is often a function of the system time. These pseudorandom number streams are a function of

a deterministic algorithm and consists of a finite sequence of numbers that appear to be random, hence they are called pseudorandom. One can imagine that all CPUs running on the same system would read the same system time and hence generate the exact same sequence of random numbers. This would of course not be ideal if we wish to simulate a stochastic system where it is assumed that random number inputs are independent. In fact, in order for each worker to have an independent stream of random numbers, the period of a random number generator should be much larger than the total number of random numbers required by all workers so there is no overlap in the random number streams. Random number generation is taken care of by the `rlecuyer` R package to ensure that each CPU has a proper stream for a given seed. As of R 2.14.0, the `parallel` package in R incorporates the L'Ecuyer-CMRG random number generator so explicit use of the `rlecuyer` package is not necessary. For the purposes of this thesis, it is assumed that non-indendence of pseudorandom number streams is properly dealt with.

A further issue arises when we wish for our results to be reproducible. Traditionally, setting the initial seed will result in the same sequence of pseudorandom numbers and hence the same results from running a simulation. However, in a parallel computing context, multiple processors will be working on the same job and even with the same initial seeds and pseudorandom number sequence, due to differences

in hardware and other external factors, different processors may work at different speeds. For example, consider a large simulation that is divided into smaller pieces and multiple processors compute these pieces one at a time on a first come first serve basis. Processors may take these pieces in a different order each time depending on the order that computation finishes. On a different system even with the same number of processors, this order that processors compute pieces in may not be the same if group of processors are faster or if the network communication is different. Hence the original results may not be reproduced. In order to account for this, one can keep track of the order in which the processors compute pieces and enforce this in subsequent simulations if one wishes to reproduce simulation results. However, if a job that was originally completed quickly by a fast processor is assigned to a slow processor in a new system, the entire job queue will be held up while this job completes. Another solution is to force synchronization so processors cannot move onto the next piece until all processors have completed, at which point chunks can be distributed in some predetermined order. This also has the issue of faster processors sitting idle waiting for all other processors to complete their computations. Neither solution will give optimal speed-up and thus reproducibility is enforced only when it desired.

2.1.3 Workload distribution

Depending on how the processors are allocated and the complexity of the problem in each sub-problem, there could be an uneven (suboptimal) distribution of workload. Some processors may be idle while other processors may be doing all of the work. Some load balancing can be done but we will introduce several ideas to minimize inefficiencies in the workload distribution. The goal is to have all processors finish their computation at roughly the same time whilst minimizing the amount of interprocessor communication required. One complicating factor in workload distribution is the uniqueness of different hardware systems since the optimal distribution of workload depends heavily on the architecture of the system that it is running on. For example, specially designed computing clusters may have extremely low interprocessor communication that is minimized through the use of specialized hardware switches while distributed clusters may have a high number of processors but intercommunication may be several orders of magnitude higher. As such, the trade off between the size of each problem and the amount of communication required between processors needs to be considered.

2.1.4 Example: Two-dimensional Heat Diffusion System

To motivate the idea of parallel programming on a lattice and to demonstrate some of the issues described above, we can consider a two-dimensional heat diffusion problem. At a given time t, the change in heat u_t is defined as a function of neighbouring values

$$u_t = a(u_{xx} + u_{yy})$$

with u_t representing the partial derivative and u_{xx} and u_{yy} representing the second partial derivatives of the temperature function $u(x, y, t)$. At time $t = 0$, the initial value at a specific coordinate (x, y) is determined by some predefined function ϕ.

$$u(x, y, 0) = \phi(x, y)$$

Using a finite difference method we can implicitly solve for the value of each lattice point $v_{i,j}$ as only a function of the value of neighbouring lattice points at the previous time step:

$$\frac{v_{i,j}^{n+1} - v_{i,j}^{n}}{\Delta t} = a\left[\frac{v_{i+1,j}^{n} - 2v_{i,j}^{n} + v_{i-1,j}^{n}}{\Delta x^2} + \frac{v_{i,j+1}^{n} - 2v_{i,j}^{n} + v_{i,j-1}^{n}}{\Delta y^2}\right]$$

$$v_{i,j}^{n+1} = v_{i,j}^{n} + a\Delta t \left[\frac{v_{i+1,j}^{n} - 2v_{i,j}^{n} + v_{i-1,j}^{n}}{\Delta x^2} + \frac{v_{i,j+1}^{n} - 2v_{i,j}^{n} + v_{i,j-1}^{n}}{\Delta y^2} \right]$$

In order to parallelize this example, we can divide the lattice into a number of sub-lattices equal to the number of processors we wish to use and distributed as such. Since initial values are not dependent on neighbours, each processor can determine the initial value of each coordinate in its sub-lattice. To compute the next time step, all the points in the lattice are traversed. If the point is an interior point, that is, all of its neighbours are fully contained within the sub-lattice, then computation can be done independently. If the point is an exterior point whose neighbour exists in another sub-lattice (not an overall boundary point), then communication is required with the adjoining processor. In fact, information needs to be swapped in both directions at once in this example as the adjoining processor will have points that require mirroring information. As one obvious optimization, the information swapping between processors can take place all at once with the entire boundary swapping in one communication step as opposed to swapping information on demand. However, since every point needs to be computes at every time step, communication must occur at every time step. An example where communication need not occur at every time step is given in chapter 4. Other discretization methods such as a hexagonal lattice can be used with potentially more accuracy but this example is just to illustrate the

communication necessary between processors in order to do the computation. In a hexagonal lattice, such communication requirements will only be exacerbated with cells having more neighbours.

2.2 Issues in Parallel Computing of Spatial Statistics

When dealing with spatial statistics and point patterns in particular, there are unique issues that arise when attempting to parallelize existing algorithms both in terms of hardware constraints, as well as mathematical considerations.

2.2.1 Memory limitations

In spatial statistics, point patterns can be represented in at least two ways: as a list of coordinates or as a rasterized grid. Depending on the sparsity of the pattern, one method may be more manageable than the other in terms of memory required to store the point pattern. Section 3.6.2 deals with large data sets that do not fit in memory.

A larger memory issue arises when computing summary statistics such as Ripley's K-function. In practice, Ripley's K-function is computed for discrete values of r

from 0 to the maximum search radius R. R can be arbitrary chosen, but is usually a function of the size of the observation window E. In order to account for edge corrections when applicable, it is far more efficient to store a matrix of distances (often Euclidean distance) from each point to all other points within R units as opposed to performing pairwise distance measures every time it is required. However, this computation requires allocating a matrix of size $n \times n$ where n is the number of points in the point pattern. As one can imagine, this does not scale well for large point patterns, although if R is relatively small, then the pairwise distance matrix can be quite sparse.

There are several ways to bypass this problem. One way to bypass the memory limitation is to randomly sub-sample from the entire point pattern to reduce the number of points we need to deal with. This will require some information to be thrown out in a trade off for a more manageable computation. In fact, multiple samples may have to be taken to ensure consistency in the measurements.

Another method to bypass the memory limitation is to use an approximate algorithm. One such algorithm utilizes a fast Fourier transform to approximate the second order K-statistic estimate done with a guard-area (border) edge correction method. This too requires throwing out information near the border of spatial windows instead of correcting for them using an unbiased correction scheme such as isotropic or

translation correction. These edge correction methods are discussed in more detail in section 3.1. A way to bypass the memory limitation without having to discard information or resort to approximations is to take advantage of parallel programming and simply scale the algorithm to work on computing clusters than can scale the memory limit linearly while decreasing computing time sub-linearly.

First-order statistics like the G, F, and J-functions are not computed based on pairwise distance measures between all points and thus do not have this memory limitation issue but improvements can still be made to take advantage of parallel computing if the data set is too large for a single computer to handle.

2.2.2 A virtual topography division

When using parallel computing, one would wish to divide a given spatial window into smaller spatial windows so that each processor only needs to deal with a manageable number of points. In single pixel algorithms where each pixel is not dependent on any other pixel, this division is trivial as there would be no communication required. However, more commonly, applications may require a processor to gather information from neighbouring windows so a virtual topography must be maintained by all processors to ensure proper communication.

The division of such spatial windows is also a matter of interest. In many cases,

the simplest division is to divide a spatial window into strips in one direction so that each processor will only need to deal with two adjacent neighbours (one for edge cases). However, in applications where communication between processors may act as a bottleneck, long narrow strips may increase the number of times we need to "cross boundaries" and perhaps even communicate with processors more than one step away if the neighbourhood radius is large. Assuming communication does not favour one direction, a better division may be one that more closely divides the spatial window into square sub-windows with points centred near the middle of these virtual divisions. That way, it may minimize the probability of a processor requiring information swapping with a neighbour.

Mineter [18] describes two methods for partitioning a lattice for extent-based algorithms, that is, algorithms that require pixels to know the state of other pixels either in a limited local neighbourhood or even globally. The first is a partitioning scheme that balances sub-window shape and number of messages. The number of messages passed is kept low by ensuring corners are shared by either 2 or 4 sub-windows and shapes are kept uniform across all processors (Figure 2.1). As a result, each sub-window will have a maximum of 4 neighbours. The second method known as heuristic partitioning is to minimize the border lengths of each sub-window which has more 'square' sub-window shape but leads to 'T' corners that are shared by 3

sub-windows, non-uniform sub-window shapes, and sub-windows with more than 4 neighbours (Figure 2.2). As can be seen, notable differences arise specifically on a prime number of processors where the topography cannot be divided into $n \times m$ grids where at least one of $n, m > 1$. For our purposes, we will maintain sub-lattice uniformity to ease coordination of communication between processors. This is generally not a big issue as one can choose a number of workers to use to create a lattice division with minimum boundary lengths (i.e., n^2 processors where n is a positive integer).

Once we get into more complex gridded topographies (more than nine processors), each processor will have up to eight adjacent neighbours – four that share an edge, and four that share a corner – all of which may be required to communicate with each other depending on how the neighbourhood structure is defined.

2.2.3 Complex spatial windows

In some applications, we may have spatial window boundaries that are complex to represent. For example, coastlines and natural or political boundaries on maps may not always consist of straight line edges. These complex spatial windows may be more easily represented as a binary mask on a lattice. In a binary mask, lattice pixels that have a majority cover in our region of interest is indicated by a 1, and

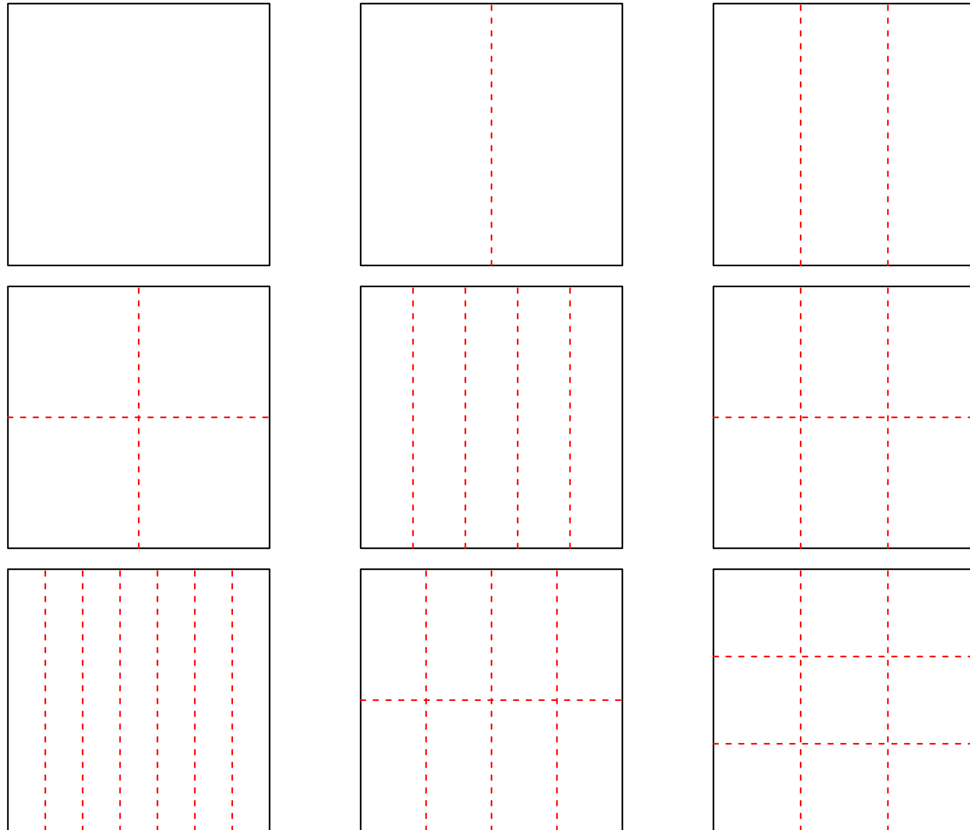

Figure 2.1: Division of lattice from 1 to 9 processors, maintaining uniform sub-lattices.

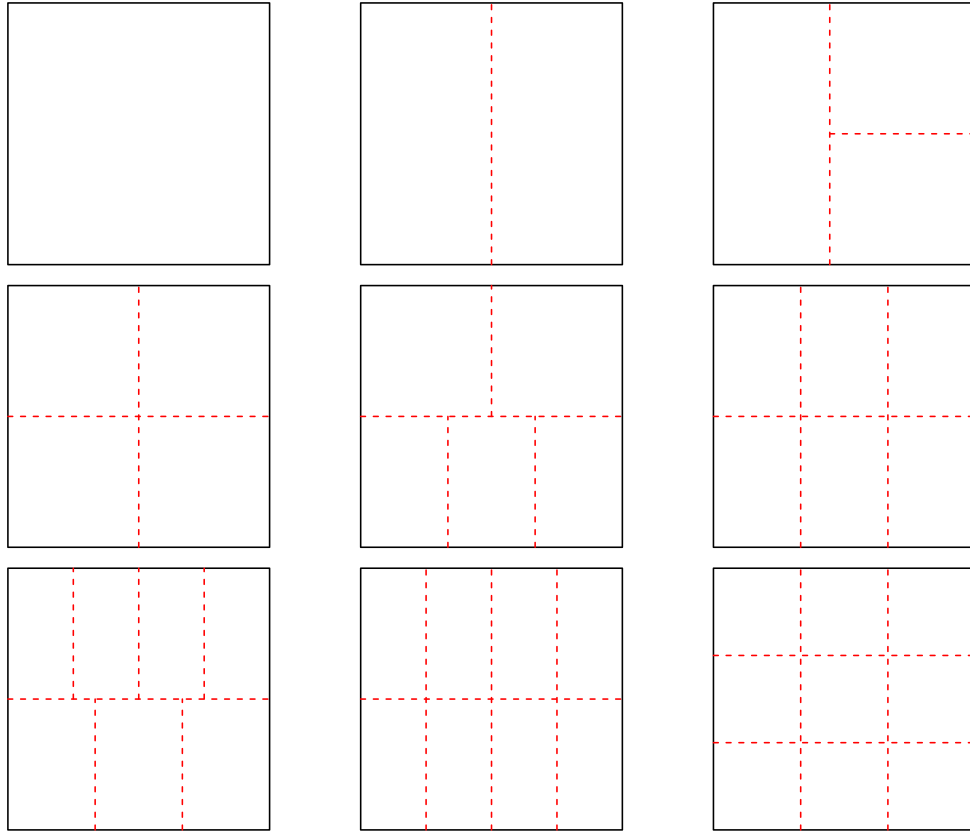

Figure 2.2: Division of lattice from 1 to 9 processors, minimizing sub-lattice boundaries.

pixels with a majority of cover outside the region is indicated by a 0. Points are also discretized in the same way depending on which pixel they fall into. Points that are close together may end up in the same pixel and are either lost or counted as a mark on the particular pixel. An obvious limitation of a discretized point pattern is the loss of information, which can be mitigated to a certain extent depending on the resolution of the lattice that is chosen. However, one should keep in mind that a larger resolution (finer detail) will result in many of the same issues that were discussed previously. A less obvious complication that may arise is when thin areas may be discretized into disjoint islands if sufficient detail is lost. Entirely enclosed islands (either natural or created through discretization) that are disjoint from the rest of the spatial window also adds another layer of complexity when parallelizing. Even assuming uniform behaviour between islands and the rest of the spatial window, special care needs to be taken to ensure that spatial window divisions account for the disjoint areas properly.

2.3 Other issues in parallel computing

2.3.1 Multiple layers of workers

Often times, we may wish to repeat a simulation multiple times to ensure consistency or for estimation purposes. An excellent example of this is estimating envelopes by simulating a random dataset many times and looking at certain quantiles (as described in section 3.4). This type of problem is embarrassingly parallel since the simulations are independent. However, if each individual simulation requires parallel processing (due to memory limitations for example), then we require parallelization at two levels. Care needs to be taken to set up the workers in such a way that each worker is aware of which manager processor it needs to communicate with. This is done using individually assigned communicators.

A map-reduce scheme is used to compile results from workers to a intermediary node. A map-reduce scheme is where a problem is 'mapped' to multiple processors for computation and then each will send results back to be reduced into a single solution. These intermediary nodes then compile their results in another map-reduce scheme to the manager processor (or depending on the complexity of the problem, another set of intermediary nodes). This type of design can scale indefinitely but the number of distinct processors required will also increase quite quickly.

2.3.2 Optimal number of processors

Communication overhead between processors adds to the overall computational time but the hope is that the amount of time saved by parallelizing far exceeds the time lost in communication and synchronization. As such, as we increase the number of processors, we get diminishing returns in computation time. For a given algorithm and hardware configuration, there is an optimal number of processors to use and even if this cannot be known exactly, knowing an approximation to this optimal number will help substantially. This optimal number is a direct function of the computation time of individual pieces. That is to say, a computationally intensive problem will scale up to a larger number of processors before the trade-off between computation time and communication is no longer worth it. Whereas a quickly computable problem (barring memory constraints) is usually better off being run on a single processor as the communication between even two processors may take longer than the computation itself. A simulation model to assess the computation time of a particular hardware set up is described later in section 2.4.

2.3.3 Load balancing

Related to the optimal number of processors, is the number of jobs to split the problem into. If we simply divide a problem into p subproblems where p is the number of processors we have access to, then uneven hardware and computational complexity will mean that each processor may spend different amounts of time to finish a computation. The final result will be limited by the processor that takes the longest to complete its job, meanwhile, the other processors that finished early would be sitting idle wasting computing cycles.

This is where load balancing comes into play. By splitting the problem into greater than p subproblems, then the first processor that finishes its computation can retrieve another job from the queue. With a fine enough job division, all workers will finish their computation at roughly the same time. However, the process of retrieving jobs from the manager and the overhead involved in setting up an individual computation may increase the overall computation time. A balance needs to be found between the number of jobs and the number of workers. A rule of thumb for the number of jobs is between 2 to 8 times the number of workers. A short simulation study can be performed to give an optimal number as a function of the expected job complexity, communication latency (hardware dependent), number of available processors, and speed of each processor.

If we have some estimate of how long a particular subproblem may take to calculate (for example, computation time may be directly proportional to the number of points involved in a computation of a summary statistic on a point pattern), then we can further optimize the load balancing by arranging jobs in order of most complex to least complex based on the point count. The idea is that the smaller jobs are able to "fill in" the leftover processing power more optimally to ensure all processors finish their computations at roughly the same time.

An inhomogenous point pattern can also have an effect on how jobs are divided. Figure 2.3 shows two point patterns, one homogenous and one inhomogenous with a higher intensity of points on the left. When the point pattern is divided into vertical slices (jobs), one expects the slices on the left to contain more points, and thus take longer to compute. In the benchmark, one can see that there is actually an increase in computation time in the inhomogenous case when the pattern is divided into too many jobs. This is because load balancing was not used and past a certain point, there is a highly irregular distribution of points amongst the jobs that cannot be reconciled without load balancing.

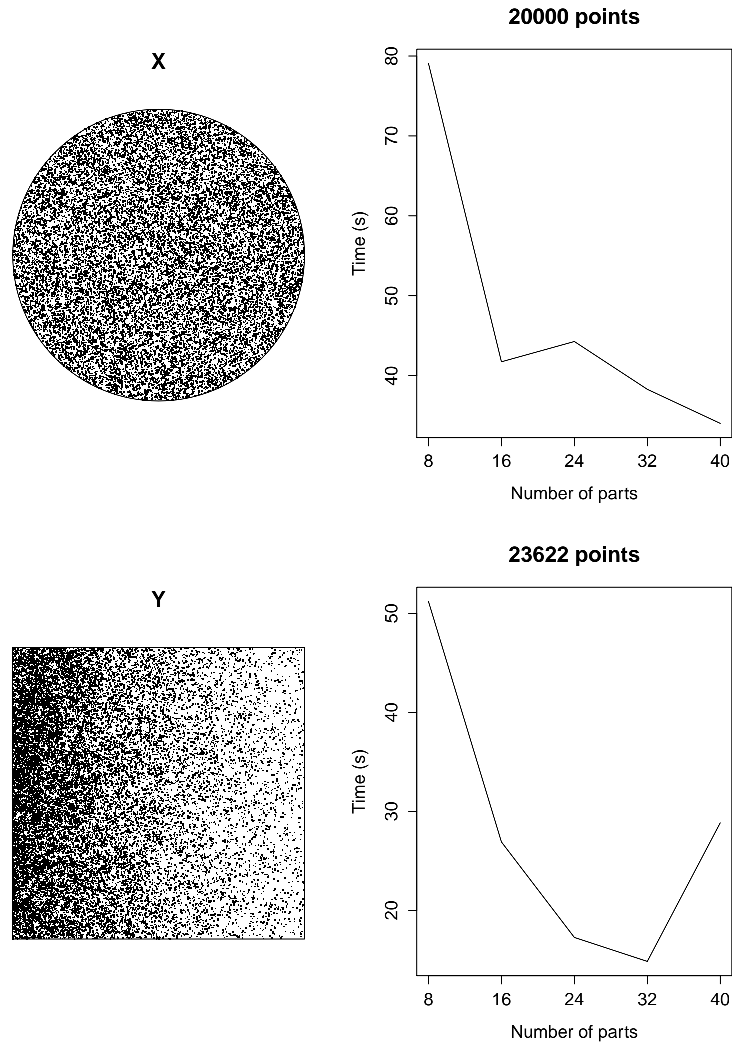

Figure 2.3: Benchmark of the number of jobs vs computation time on two different point patterns.

2.4 A simulation model of optimal parallel computing structure

All of the aforementioned issues can be incorporated into a single statistical simulation model with processing and communication times represented by various distributions. For different hardware configurations and different problems, we may have a good estimate of the parameters for these distributions to an order of magnitude. In a distributed computing model over a slow network connection (e.g. the Internet), the communication time may be on the order of seconds. Whereas on a high performance computing cluster, communication time may only be on the order of several milliseconds.

2.4.1 A simple model

We can first define some variables,

- S = the maximum number of workers available
- J = the number of jobs our problem is divided into
- d = the size of the data
- s_i = the status of the worker i indicating the time until completion of its job

- t_c = constant overhead of transmission (unavoidable communication time)
- c_c = constant overhead of computation (problem set up independent of data)

For each job $i = 1, ..., J$, both the communication overhead, t_i, and the computational time, c_i, can be assumed to increase linearly with the size of the dataset, d. They can be represented as:

$$t_i \sim N(t_c + f(\frac{d}{J}), \sigma_t)$$

$$c_i \sim N(c_c + f(\frac{d}{J}), \sigma_c)$$

where σ_t and σ_c are the standard deviations in communication and computation time, which can be approximated through small experimental trial runs of a particular problem (these are not dependent on the number of workers used, S, or the size of the data, d). $f(x)$ is a function that determines the time to perform the necessary computations on data of size x and can be estimated or measured empirically on a given hardware setup as it will be highly dependent on processor speed.

The following algorithm can then compute the overall time to complete the entire job using S number of workers.

Step 1. Set all workers statuses $s_1, \ldots, s_S$ to 0; $j = 1$

Step 2. For job j, find $s_{min} = \mathbf{min}(s_1, ..., s_S)$

Step 3. Set $time = time + t_j$

Step 4. If $time < s_{min}$ then $s_{min} = t_j + c_j$; otherwise, $s_{min} = time + c_j$

Step 5. $j = j + 1$ and go to Step 2; stop when $i = J$

Note that it is assumed that the final transmission time to gather all the results in the manager is negligible as it will simply be a summary statistic or function that is relatively constant in size regardless of the size of the original data.

2.4.2 A simple model with concurrent job distribution and load balancing

The model above can be improved by allowing for jobs to be distributed concurrently (especially important during the initial distribution of jobs where all workers are waiting to begin) and also by incorporating a simple form of load balancing based on job complexity (represented by computation time $c_{1...j}$).

The algorithm is then modified to the following:

Step 1. Order $t_{1...J}$ and $c_{1...J}$ in decreasing order of $c_{1...J}$

Step 2. Set all workers statuses $s_1, \ldots, s_S$ to 0

Step 3. Set $s_i = t_i + c_i$ for $i = 1, ..., S$; $time = \mathbf{max}(s_1, ..., s_S)$; $j = S + 1$

Step 4. For job j, find $s_{min} = \mathbf{min}(s_1, ..., s_S)$

Step 5. Set $time = time + t_j$

Step 6. If $time < s_{min}$ then $s_{min} = t_j + c_j$; otherwise, $s_{min} = time + c_j$

Step 7. $j = j + 1$ and go to Step 4; stop when $i = J$

Using such a model, optimal values for the parameters S and J can be found under given constraints of hardware implementation in order to minimize overall computation time. In practice, J should be at least twice as large as S.

In order to assess the effect of load balancing with varying number of jobs, a simulation was carried out assuming a single node with 8 processors. This was done to ensure negligible communication time between processors since they are all on the same node. This simulation was repeated for an increasing number of jobs and with and without load balancing. Results of the simulation can be seen in Figure 2.4. It can be seen that use of load balancing (red line) has an improvement over the same calculation done without load balancing (black line). This effect is more evident when the number of jobs is large but as can be seen in the third graph, its performance improvement is limited when the computation is expensive enough that computational time of jobs are more or less the same. Past a certain point, increasing the number of jobs does not increase the speed gain.

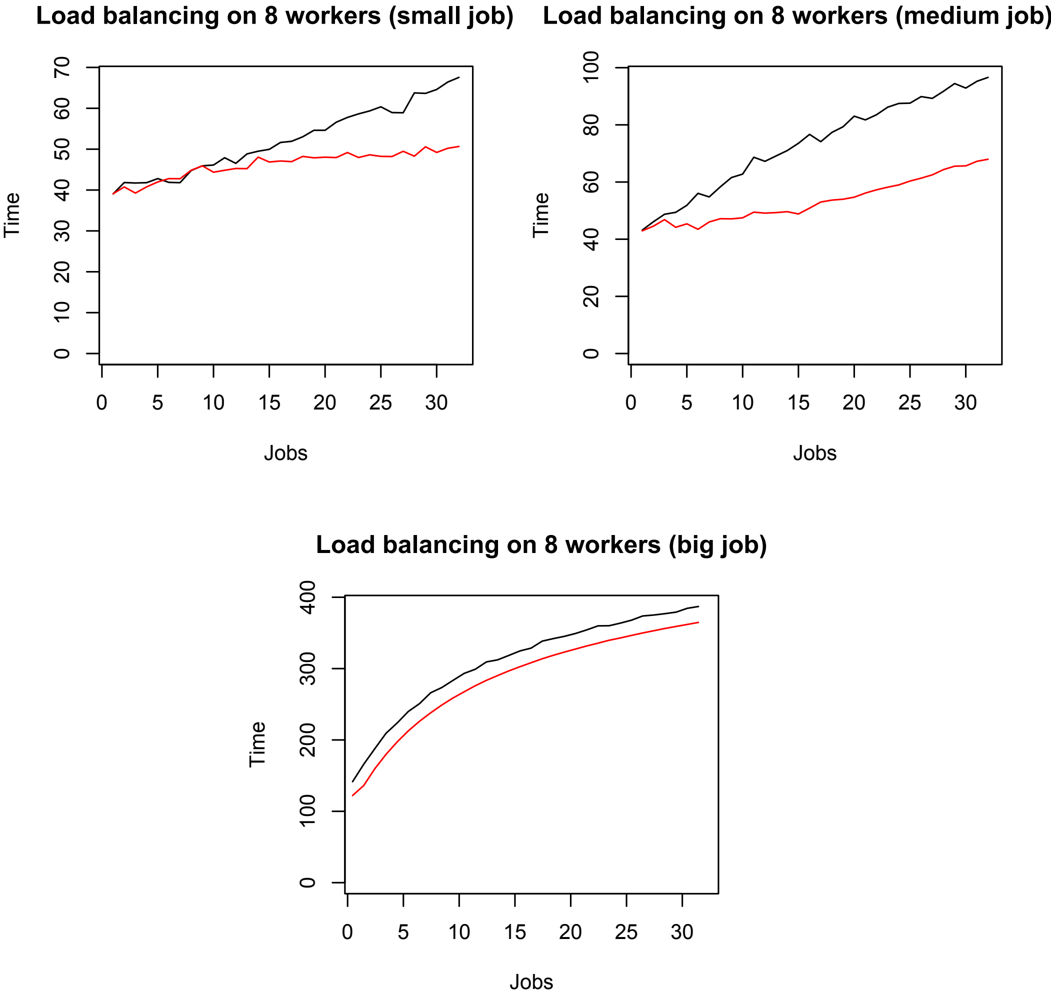

Figure 2.4: Benchmark of the number of jobs (in excess of the number of workers) vs time with load balancing (red line) and without load balancing (black line) for varying data sizes.

Chapter 3

Parallel computing for spatial point processes

3.1 Edge correction methods

Before discussing parallel computation of spatial summary statistics, we will first discuss edge correction methods as it is these methods that create difficulty in parallelizing computations. Edge correction methods to create an unbiased estimator can be divided into two types, one that excludes points for which we do not have full information, and ones that apply a weight to pairwise distances of points to account for points outside of the spatial window. Several such correction methods

of each type are described below. If we are only interested in computing a naive estimator of a summary function or only interested in a simple correction method that excludes points, then the computation can be embarrassingly parallelized as it does not matter where a point is located in a spatial window since edge effects are essentially ignored. For summary characteristics that depend on the actual location of each point, the following explanations of edge correction shown on the K estimate are applied similarly.

Toroidal correction

For simple spatial windows like a rectangle, toroidal (periodic) edge correction can be applied where the edge on one side can be thought as wrapping around to the opposite edge. Unless one expects the behaviour of points on one side of the window to be the same as the other side (i.e., a two dimensional representation of a curved toroidal surface), the K estimate may not be physically valid or sensible. The implementation involves replicating the point patterns in the spatial window eight times, surrounding the original window and calculating the K estimate only treating the original points as centers upon which to count surrounding points (Figure 3.1). Each point in the spatial window is replicated 9 times (including the original point pattern) and the interpoint distance from a point i to some other point j is the shortest of all distances

from point i to the 9 replicates of j.

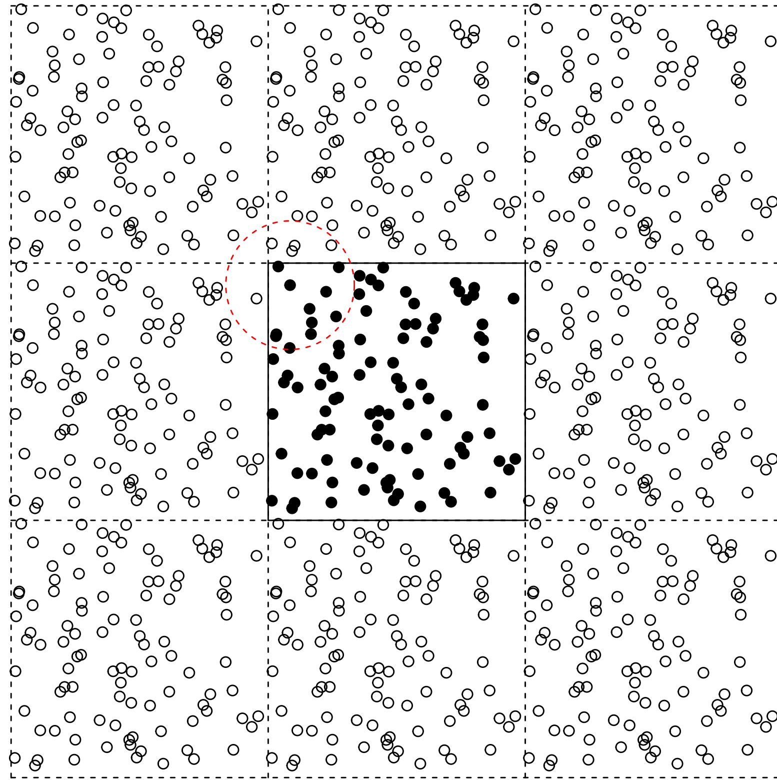

Figure 3.1: Toroidal edge correction on a point pattern (solid points) by replicating the point pattern 8 times (hollow points) but only treating the original points as centers.

Related to this correction is reflection correction where instead of replicating the same point pattern around the original spatial window, the surrounding spatial windows are reflected along the edge or corner of intersection instead. This reflection correction will amplify behaviour of points at edges as edges with many points will

be reflected to have more close neighbours while an edge without a lot of points will not have many. As one can expect, assuming the toroidal or reflection assumption is valid, these method work best for rectangular spatial windows where the geometry lends itself to this replication scheme. For more complex spatial window shapes, we can resort to other edge correction methods.

Border method

Perhaps the most straightforward edge correction method for an arbitrary window is when one actually has the neighbouring point information for points that are within some maximum search radius R from the border. This is known as plus sampling but it is rarely the case that we are able to get this information. Instead, we can redefine our existing point pattern by reducing the effective spatial window so that only points more distant than R from the perimeter of E are considered in the analysis. The other points outside the reduced spatial window can then be considered the neighbouring point information. These ignored points are still included in the count for interior points beyond r from the edge, but are not considered centre points themselves (Figure 3.2). This method is known as the border method, the guard area method, buffer zones, or minus sampling. It can theoretically work for windows of any shape given a reasonably small search radius R but too large a search radius

may result in disjoint interior windows (Figure 3.3). By ignoring all points within R distance of all edges for the purposes of computing the K estimate, we can guarantee that edge effects will not be introduced. However, depending on the density of points near the perimeter of the window, this may result in much of the dataset being removed, more so for large values of r. Sterner et al. [40] found that by creating a buffer zone at the largest radius that they wanted to analyze, they essentially required up to four times the size of the original data to have been recorded to keep the desired spatial window the same.

The formalization of the border method (for the K-function) is given as,

$$\hat{K}_{border}(r) = \frac{E(n_o(r))}{\lambda} = \frac{1}{\lambda m}\sum_{i=1}^{m} n_i(r) \text{ for } r \geq 0.$$

where m is the number of points of N in $E \ominus b(o,r)$, the reduced spatial window formed by Minkowski subtraction of $b(o,r)$ from the full spatial window E.

The border method is traditionally the only edge correction that is computationally efficient enough to be used on large datasets since it is essentially computing a naive estimate on a subset of the point pattern without having to do any actual edge correction calculations. In order to speed up this calculation further, the **spatstat** package [3] recommends that a fast Fourier transform (FFT) method be used as an

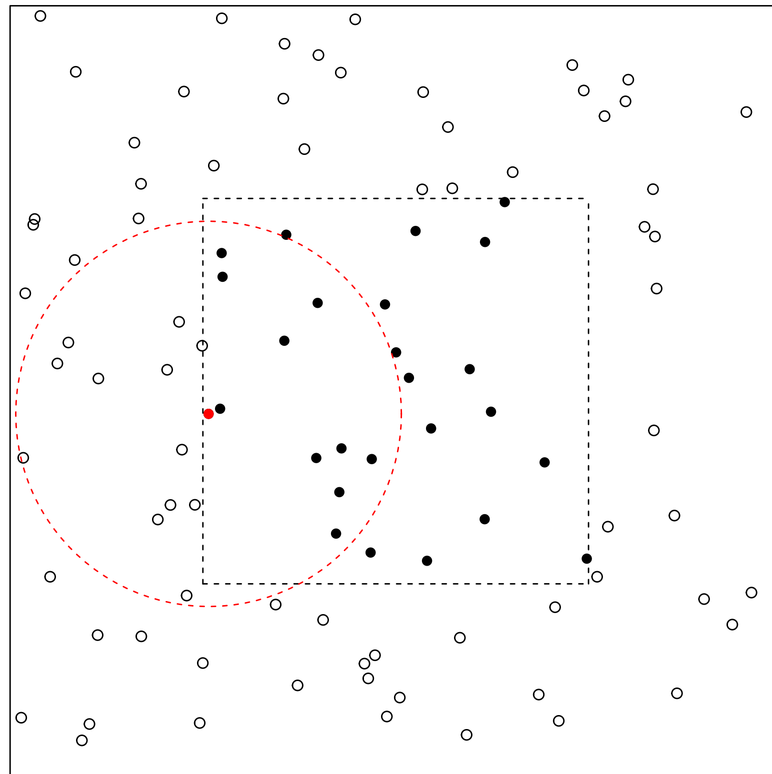

Figure 3.2: Border method edge correction on a point pattern by discarding points within some maximum search radius from an edge (hollow points). Only interior points (solid points) are used as centers. In this example, it resulted in 74% of points being discarded using a maximum search radius of 1/4 of the square window dimensions.

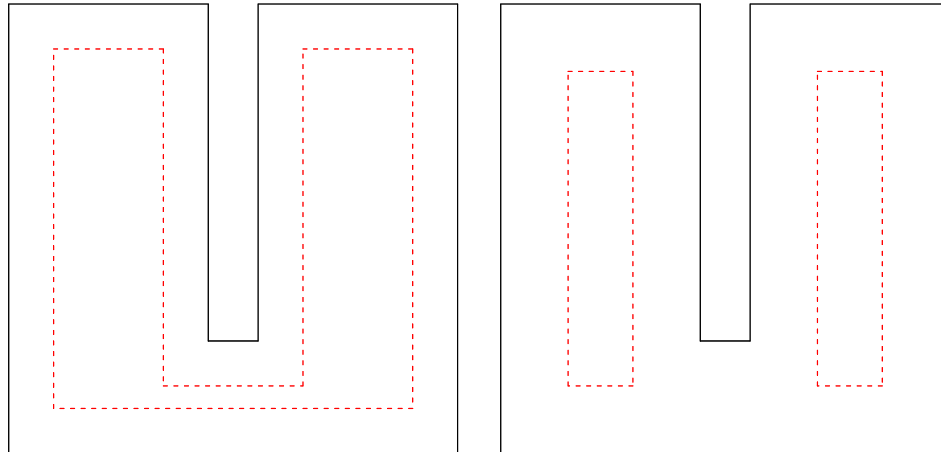

Figure 3.3: The border method may not work well for even a simple window shape if it needs to be reduced by the some maximum search radius (left). It may result in disjoint interior windows if the maximum search radius is too large (right).

approximation to the K-function with the border method. This approximation is done by first creating what is known as a Fry plot by translating each point along with the relative location of all other points to the origin. The result is a plot that gives the total number of points within a certain distance d of all points by simply looking at a circle of radius d centred around the origin. An example using the *cells* dataset from the **spatstat** package in R and the corresponding Fry plot is given in Figure 3.4. This dataset contains the locations of 42 cells observed under a microscope with each point representing the centre of each cell. The shape and size of the cell naturally produces an inhibition pattern. The fast Fourier transform is utilized

to smooth this Fry plot in order to quickly get smoothed estimates of the reduced second moment function, which can then be used to get an estimate of the K function (Figure 3.5).

Although the border method can be computed quickly for large datasets using the aforementioned approximation, it would be interesting to examine how good these approximations are compared to the exact K estimate. The **parspatstat** package described in this paper will allow us to compute the exact K estimate for large datasets quickly. More interesting however, are the isotropic and translation edge correction methods that use all the points available. It is not computationally feasible (without taking advantage of parallel computing) to compute the K estimate exactly for large datasets using these two edge correction methods and the above approximate method cannot be applied here. There are other edge correction methods as well that are not described here but are supported by **spatstat** and hence **parspatstat**.

Isotropic correction

Ripley's original proposed edge correction, known as isotropic correction or simply Ripley's correction, adjusts the naive K estimate by scaling (weighing) it by the ratio of the circumference of the search circle that lies inside the window to the circumference of the entire search circle. Explicit formulas can be given for simple

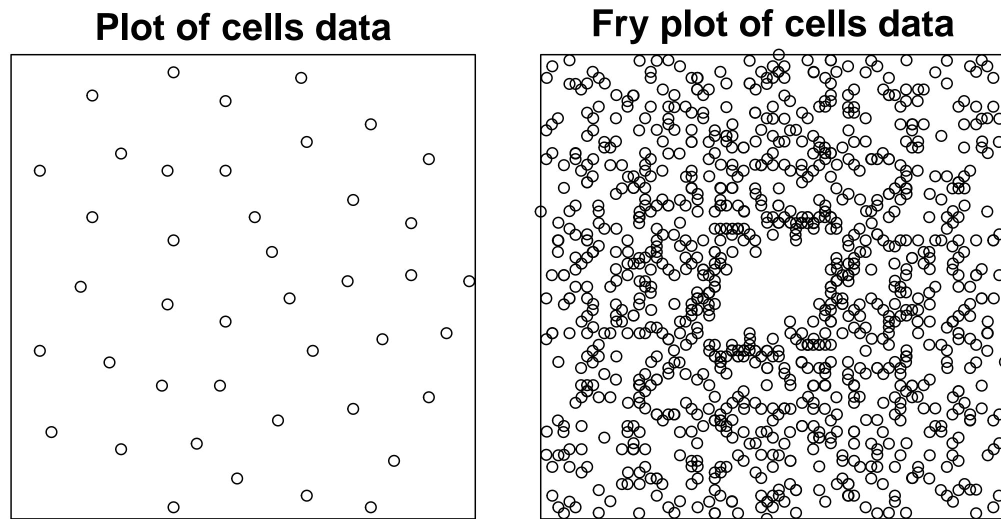

Figure 3.4: The plot of *cells* data (left) and its corresponding Fry plot (right).

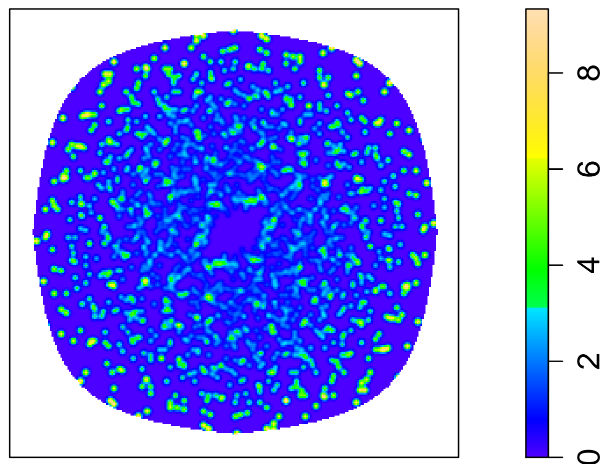

Figure 3.5: Smoothed estimates of the reduced second moment measure, from which an estimate of the K function can be obtained at varying radius distances from the origin.

rectangular or circular windows [13] but can be difficult to compute for non-standard window shapes. Also, this correction assumes isotropy of points (our search area is a circle as opposed to an ellipse), hence the name isotropic correction. If the degree of anisotropy is known, one could first perform an appropriate projection of the points and use the same method.

The weight assigned to pairs of points can be written as

$$w(x_1, x_2) = \frac{|\delta b(x_1, ||x_1 - x_2||)|}{|\delta b(x_1, ||x_1 - x_2||) \cap E|}$$

where $|\delta z|$ is a measure of the perimeter of window z so $|\delta b(x_1, ||x_1 - x_2||) \cap E|$ is the circumference of the search window that lies inside the spatial window E.

Translation correction

Another common edge correction method is translation correction [25]. Similar to the isotropic correction, the naive K estimate is adjusted by a scaling factor (weight). The scaling factor in translation correction is the area of intersection between the original E and translated window $E_{x_2 - x_1}$ that is created by translating the window in the direction and distance of a point x_2 to another point x_1. One advantage of the translation correction method is that it can be applied to windows of any shape,

though it can be more computationally intensive for windows of non-standard shape.

Here, the weights assigned to pairs of points can be written as

$$w(x_1, x_2) = A(E_{x_1} \cap E_{x_2}) = A(E \cap E_{x_1 - x_2})$$

where $E_x = \{z + x : z \in E\}$ is the spatial window E entirely translated by x so $A(E_{x_1} \cap E_{x_2})$ is the area of overlap between the two translated windows.

3.1.1 Comparison of edge correction methods

Haase [17] summarizes and carries out a comparison of the different methods of edge correction applied to Ripley's K-function in which he uses an experimental data set and a real data set. In the real data set, the use of the border method edge correction could not be performed because of insufficient field data with the author noting that an area up to four times the analyzed plot is potentially left out of the analysis as considered points.

Yamada and Rogerson in 2010 conducted an empirical comparison of the different edge correction methods for the K-function and found that isotropic correction and toroidal correction was more powerful than the border method or not using any correction at all [49]. To further extend these findings, an empirical comparison of

the edge correction methods with relation to their variance was done.

Although the border method is computationally efficient, the exclusion of potentially important points near the boundaries causes it to not be as useful as isotropic or translation correction. Also, although the three aforementioned edge correction methods (we are not considering toroidal correction) are all unbiased estimators, the variance of the border method estimate is higher than that of isotropic and translation correction. This is demonstrated by a repeated simulation of a constant intensity Poisson process with $\lambda = \{100, 500, 1000, 3000\}$ in a disc with constant parameters and having each edge correction method applied to the K estimate on each point pattern. The results are shown in Figure 3.6 and confirm what is shown by Ripley in [34].

3.2 Parallel computation of point process summary functions

A general idea in spatial statistics is that points that are close together may be related and this relationship diminishes as points are farther apart. Spatial summary statistics therefore, are often local computations in that they are only concerned with points that are close to one another. This is obvious with nearest neighbour statistics

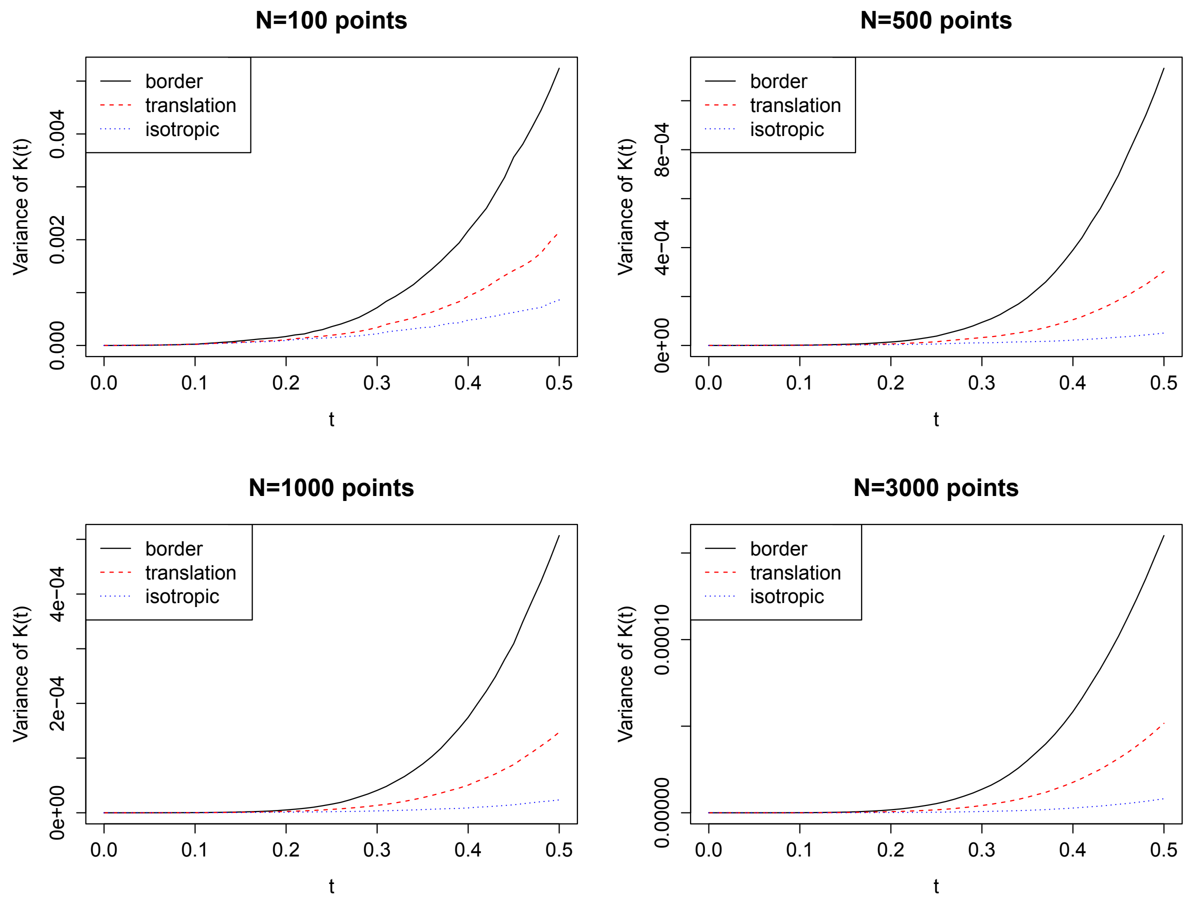

Figure 3.6: Estimated variances from simulation of binomial processes with varying n under border, isotropic, and translation correction methods.

such as the empty space function. In spatial summary functions like the K or pair correlation function, pairwise points are only considered up to some maximum radius that is a fraction of the spatial extent of a point pattern. Although summary functions can be considered with a global extent, there is little information gained as compared to a more reasonable local extent.

We can take advantage of this local calculation to incorporate parallel computing. The basic idea is that since the calculation of a statistic on an arbitrary point only involves a small fraction of other points near it, workers do not need to know the entirety of the point pattern. This reduces memory requirements while speeding up computation. The speed up in computation is possible when individual computation on a fraction of the points can be combined with other similar computation to get the exact answer as what one would expect from a calculation on the whole pattern. We can illustrate this through a parallel estimation of the naive K-function (without edge correction).

Given a point pattern on a square spatial window, we can divide the spatial window into two sub-windows (Figure 3.7). Although each processor is only responsible for a single sub-window, it requires knowledge of points that are outside of the sub-window but are within the maximum search radius of a boundary.

A similar argument can be made with an unbiased edge corrected K-estimates.

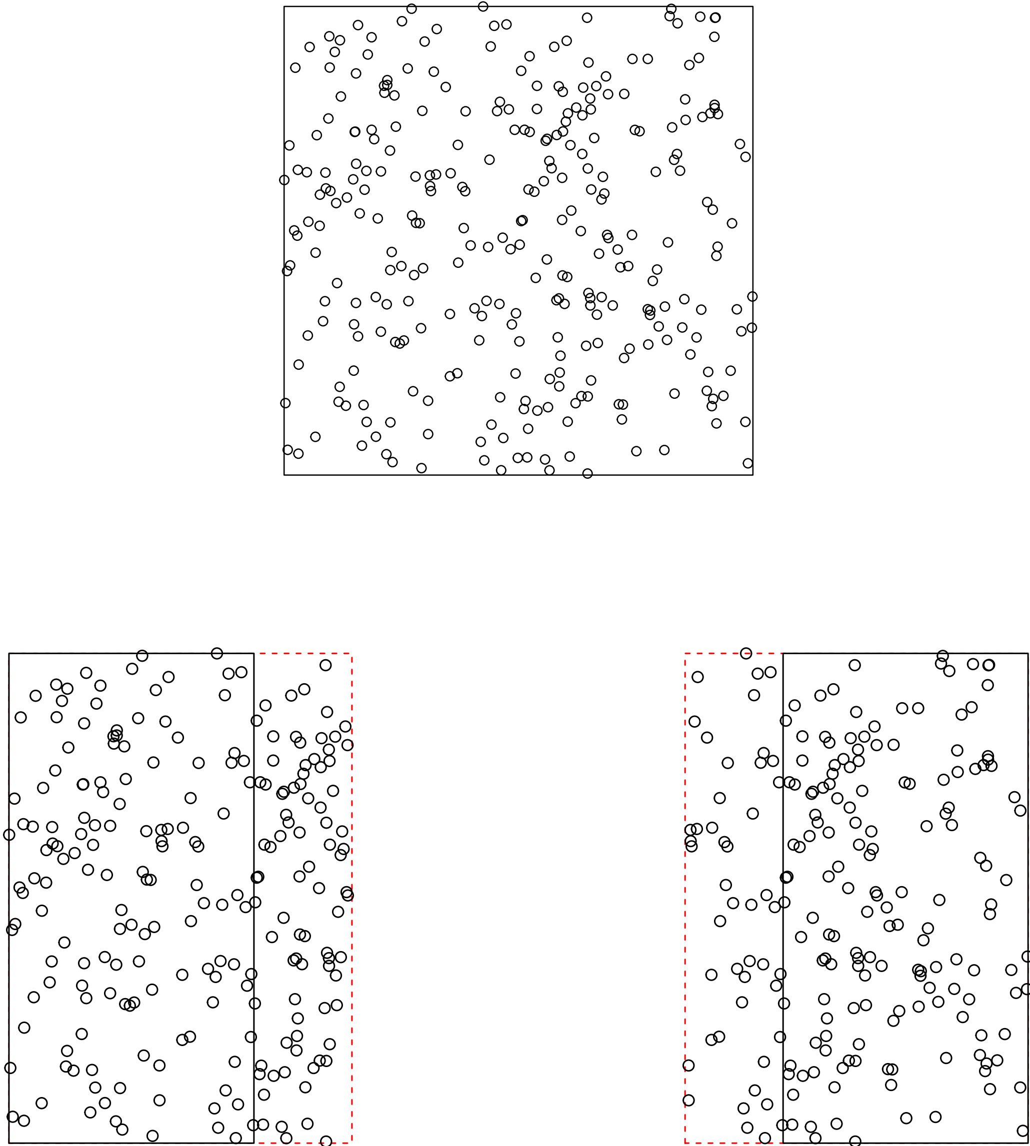

Figure 3.7: A simple example of dividing a point pattern (above) into two sub-windows (below). Although the spatial window is divided by half, points that are within the area covered by the maximum search radius (area enclosed by dotted red line) still need to be stored.

The difference is that each sub-window needs to be aware of where it lies in the entire spatial extent. In other words, each sub-window needs to be aware of which edge is a true edge and which is simply adjacent to another sub-window in the larger spatial window. Edge correction should only apply to those edges that are true edges. For example, in a division of a square lattice into three sub-problems (Figure 3.8), the middle slice has two interior boundaries for which edge correction should not be used, while the outer pieces only have one such interior boundary.

Figure 3.8: When applying edge correction in parallel computation of spatial statistics, each processor is aware of only the extent of its own sub-window, yet needs to be aware of whether a boundary is a global boundary (black) and requires edge correction, or an interior boundary (red) that requires no edge correction.

3.3 Performance benchmarks

Actual benchmarks were computed to assess the speed-up of computing a summary statistic (the K-function) in parallel as a function of the number of processors used on point patterns of various sizes. A homogenous Poisson pattern was created with $n = \{5000, 10000, 25000, 50000\}$ points and the `parKest` function from the accompanying `parspatstat` package was used to compute the K-estimate using varying numbers of processors. The results are displayed in Figure 3.9 and it can be seen that there is a sharp decrease for using several processors in parallel but there are diminishing returns as communication overhead begins to overtake any gains in computational speed. Computational speed-up is measured as the fraction of the time it takes to perform the computation on just a single processor to the time it takes to do the same computation on multiple processors. The speed-up for various point pattern sizes is shown in Figure 3.10 which more clearly shows the diminishing returns as well as the actual speed decrease from using too many processors in parallel and introducing communication overhead.

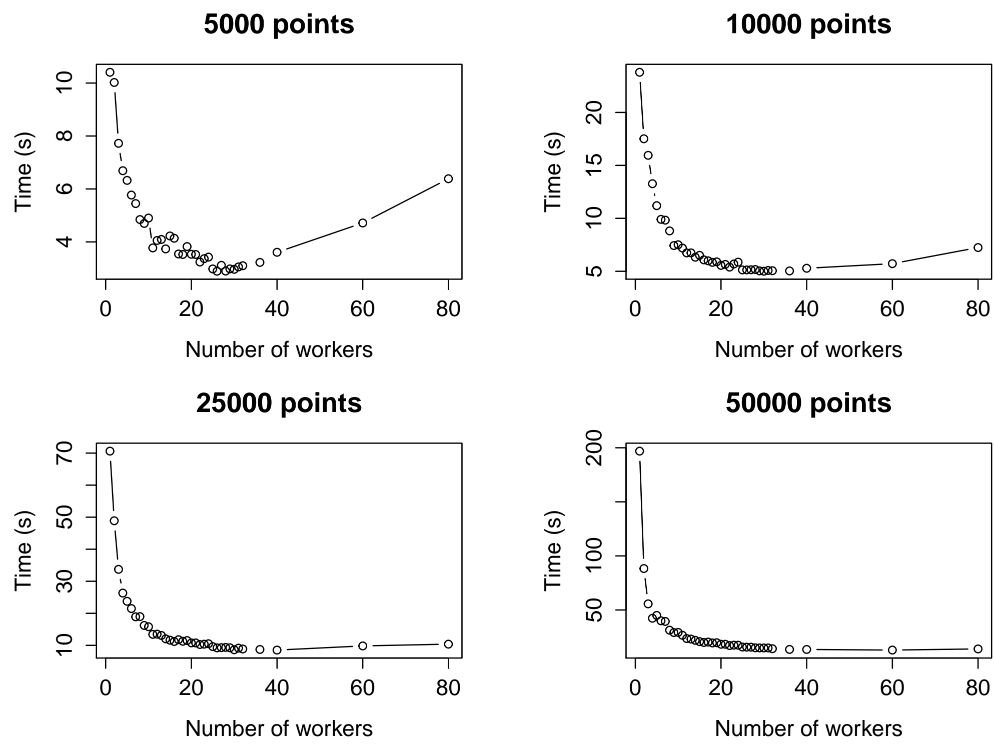

Figure 3.9: Benchmark of the number of CPU vs time for uniform point process patterns of varying size.

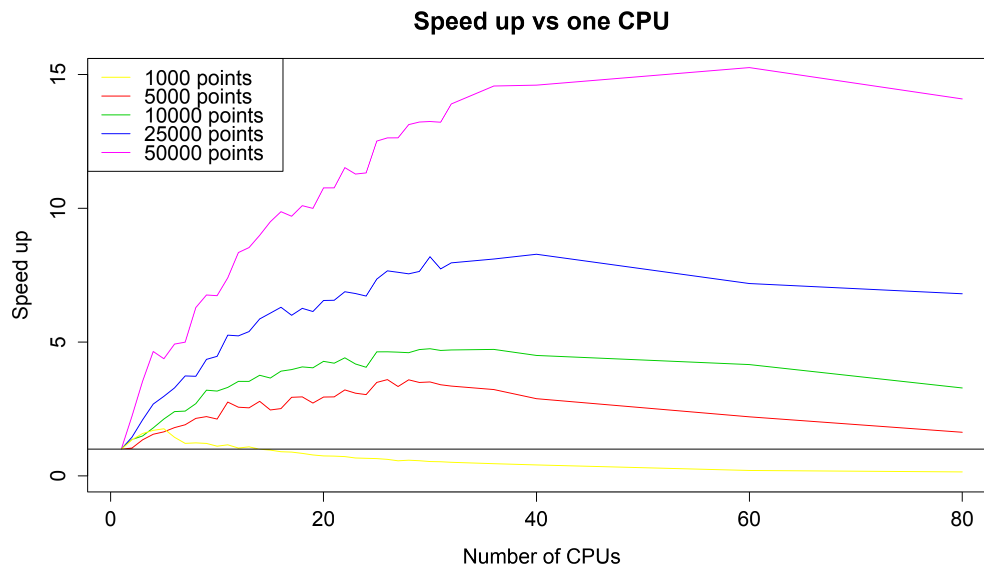

Figure 3.10: Speed up of parallel K-estimate for point patterns of varying sizes.

3.4 Point process model fitting and evaluation

Estimating error using bootstrap resampling has been examined thoroughly by Efron and Tibshirani [12]. The idea is to resample a new sample of the same size from our original data with replacement.

As pointed out by Martinez and Saar [24], this type of bootstrap error estimate is not valid when dealing with point process statistics for several reasons. First, all samples of the same size as the original point pattern when sampled with replacement will always contain at least one duplicate point, unless the sample is equivalent to the original point pattern. On average, about one third of the sample will be dupli-

cated [24]. This introduces a new feature (duplicate points) that did not exist in our original point pattern, and any statistics calculated from a sample would be invalid. For example, nearest neighbour statistics will include points whose nearest neighbour is in the same location as itself. Discarding these points will result in a sample that has a much lower point density (intensity) than the original sample, again invalidating any statistics calculated from it.

The resulting bootstrap variance from the procedure as described above has in fact been derived analytically [38] and is not the variance of the point process that we are interested in. In order to estimate the bootstrap variance of a point process, the bootstrap samples should be new whole samples obtained from carrying out the survey again [22]. A parametric bootstrap may be more suitable where estimated parameters are used to generate realizations from a model and new parameter estimates are made based on these new realizations. The resulting parameter estimates can then be used as an analogue to the variance of the original parameter estimator. A similar approach can be used to assess the fit of a point process model by repeated sampling from the model itself using simulation envelopes.

3.4.1 Simulation envelopes

Simulation envelopes can be used to assess the fit of a point process model or to test a point pattern to see if it differs from a complete spatially random process or some other point process model. An envelope is created by simulating a point pattern following a specified point process model N times, each with the same parameters, such as intensity, as the original point pattern and with the same spatial window extent. K-estimates (or some other summary statistic on which to compare) are computed for each of the N realizations and an envelope is created by taking upper and lower quantiles (i.e. the maximum and minimum). The envelope is usually computed point-wise (taking upper and lower quantiles at each value r) but can also be computed globally where entire curves are considered for quantile instead. The envelope can also be created by assuming a normally distributed summary statistic or be computed as a certain number of sample standard deviations from the sample mean. If the observed point pattern falls within the simulated envelope, then there is no evidence that the point process model generating the envelope fails to be a good description of the point pattern, though this should not be interpreted as a confidence interval [33].

Subsequent realizations of the simulation envelope will differ, with the variance of the quantiles dependent on the number of simulations run to generate the enve-

lope. More simulations should be run to reduce this envelope variance. This is an embarrassingly parallel problem as each simulation is independent of others and thus can be combined. A parallelized version of the `envelope` function from `spatstat` is available in `parspatstat` in the `parenvelope` function.

3.5 Point process reconstruction

The idea of reconstructing a point process based on one or more of its summary statistics and summary functions was proposed by Tscheschel and Stoyan in 2006 [46]. The basic idea is to simulate a new point pattern that is similar to an existing point pattern in one or more predefined characteristics.

There are several situations where one might be interested in reconstructing a point process pattern based on its summary characteristics (summary statistics and summary functions). For example, in forestry, it is often expensive or impractical to identify the location of every tree in a forest. Instead, a common technique used is to pick trees in a small plot and measure nearest neighbour summary statistics [28]. From these statistics, one can use a reconstruction algorithm to reproduce a point pattern in a larger area with matching characteristics. Such a pattern is not expected to reflect the true position of trees, but for example, may serve as a proxy for a point

pattern if a large area of trees is required.

Another example is when a point pattern was sampled entirely in a spatial window, but one would like to examine the behaviour of the point pattern with different window shapes and sizes. The point pattern can be reconstructed within a larger enclosing spatial window conditional on the points within the original window staying the same.

Finally, reconstructed point patterns can provide input to simulation models or even to evaluate the summary characteristics as was done by Pommerening [27] with the aim of finding out which characteristics describe the distribution of trees in forests particularly well [20].

3.5.1 The general reconstruction algorithm

The general idea is to start with a random point pattern and perturb points one at a time, keeping new point patterns that are closer in summary characteristics to the desired point pattern. We will use the same notation as described in the original paper by Tscheschel and Stoyan [46]. The observed point pattern is denoted φ and the observed spatial window is denoted W_{obs}. A reconstructed point pattern is denoted ψ and the simulation spatial window is denoted W.

As a minimum constraint, we will hold the intensity λ constant, that is to say,

the number of reconstructed points n in W is known exactly and in the case where $W = W_{\text{obs}}$, n is the number of points in the observed point pattern.

Let

$$n_i \text{ for } i = 1, \ldots, I$$

$$f_j(r) \text{ for } j = 1, \ldots, J$$

be predetermined summary characteristics on which we wish the reconstructed point pattern to match the observed point pattern. $\tilde{n}_i(\varphi)$ and $\tilde{f}_j(r;\varphi)$ are the estimates of n_i and $f_j(r)$ on the observed point pattern φ. The target is to construct a point pattern with n points and estimated summary characteristics $\hat{n}_i(\psi)$ and $\hat{f}_j(r;\psi)$ close to $\tilde{n}_i(\varphi)$ and $\tilde{f}_j(r;\varphi)$. That is,

$$\hat{n}_i(\psi) \approx \tilde{n}_i(\varphi) \text{ for } i = 1, \ldots, I.$$

$$\hat{f}_j(r;\psi) \approx \tilde{f}_j(r;\varphi) \text{ for } r \in [0, R_j] \text{ and } j = 1, \ldots, J.$$

where R_j is the maximum search radius for the j summary functions.

In order to achieve this, we need a way to measure how close two summary characteristics are. This is defined by an 'energy' function, $E(\psi)$, which measures the deviation of a summary characteristic on the simulated point pattern to the observed

pattern.

For a summary statistic, $E_{n_i}(\psi)$ is defined as

$$E_{n_i}(\psi) = [\tilde{n}_i(\varphi) - \hat{n}_i(\psi)]^2 \text{ for } i = 1, \ldots, I.$$

and for a summary function, $E_{f_j}(\psi)$ is defined as

$$E_{f_j}(\psi) = \int_0^{R_j} [\tilde{f}_j(r;\varphi) - \hat{f}_j(r;\psi)]^2 \mathrm{d}r \text{ for } j = 1, \ldots, J.$$

These energies for different summary characteristics can then be combined into a single value $E(\psi)$ where

$$E(\psi) = \sum_{i=1}^{I} E_{n_i}(\psi) + \sum_{j=1}^{J} E_{f_j}(\psi).$$

General reconstruction algorithm

Given: An observed point pattern φ in W_{obs}, summary characteristics $\tilde{n}_i(\varphi)$ and $\tilde{f}_j(r;\varphi)$, energy change threshold $\varepsilon > 0$, maximum total iterations S, and a maximum number of iterations without significant change, m.

Step 1. Generate a random point pattern ψ_1 from a binomial process with n points where $n = \lfloor \lambda * A(W) + 0.5 \rfloor$.

Step 2. Set $s = 1$

Step 3. do while $E(\psi_{s-m}) - E(\psi_s) < \varepsilon$ and $s < S$

(a) Create ψ'_s by perturbing a random point in ψ_s to a random location in W.

(b) Update ψ_{s+1}

$$\psi_{s+1} = \begin{cases} \psi'_s \text{ if } E(\psi'_s) \leq E(\psi_s), \\ \\ \psi_s \text{ otherwise.} \end{cases}$$

(c) Set $s = s + 1$.

Step 4. ψ_s is a reconstruction of φ.

Figure 3.11 shows an observed cluster pattern and the reconstructed pattern matching on the K-function with 1,500 iterations and the corresponding decrease of the energy function.

It should be reiterated that summary characteristics do not uniquely identify a point process pattern or even a point process model. For example, it was shown by Baddeley et al. that an artificial random cluster process can produce the same second-order characteristics as a Poisson process [4]. However, if a point pattern is reconstructed by matching summary characteristics that are of importance to the application, then this provides an adequate reconstruction within the scope of the

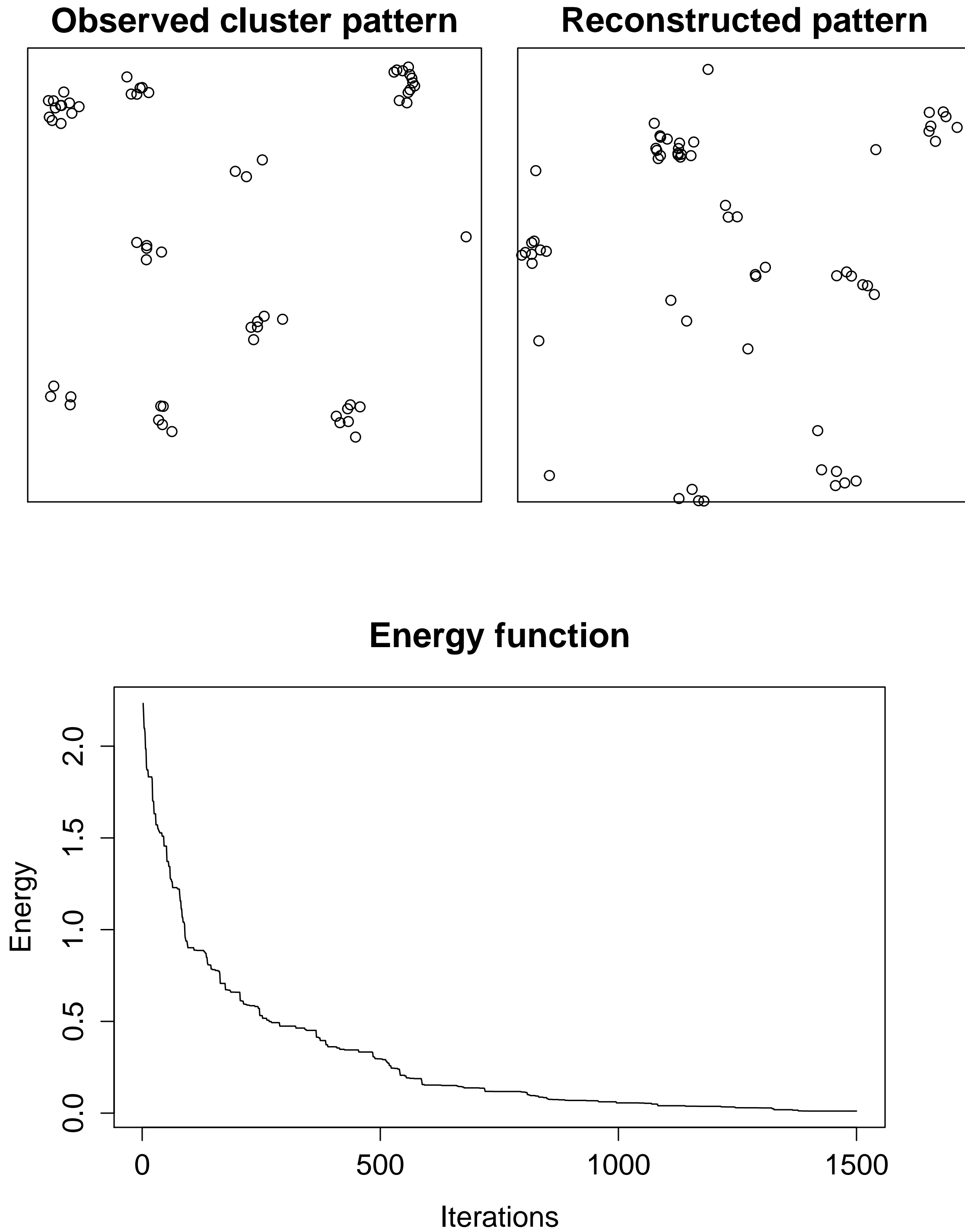

Figure 3.11: The observed point pattern and a reconstructed point pattern (top). Energy function of the reconstructed point pattern over 1,500 iterations (bottom).

specific problem.

When selecting the characteristics to use, it is often wise to select characteristics that measure different things. An example of poor characteristic selection would be to select the K- and L-function, as they contain equivalent information. Also, characteristics cannot be contradictory, for example, a targeted K-function indicating clustering and a targeted L-function indicating inhibition. This generally won't be an issue if the target summary characteristics are taken from an actual observed point pattern as they will automatically be internally consistent but if summary functions are proposed arbitrarily, there may not exist a point process that satisfies the proposed summary characteristics.

Tscheschel and Stoyan [46] also noted a variation of the algorithm using ideas from simulated annealing by Kirkpatrick et al. [23] to attempt to find a global minima for $E(\psi)$ as the basic algorithm may get stuck in a local minima. The idea with simulated annealing is to give a probability for updating $\psi_{s+1} = \psi'_s$ with some probability even when $E(\psi'_s) > E(\psi_s)$. This probability decreases as s increases to mimic the physical metallurgy process of annealing where large changes are made when the temperature is hot and the magnitude of these changes decreases as a function of the decreasing temperature over time. Pommerening and Stoyan [28] also introduce a hardcore distance constraint on points by only accepting a new point if it is not within a

certain distance of another point. Even with these adjustments, this reconstruction algorithm has some limitations.

3.5.2 An improved reconstruction algorithm

The reconstruction algorithm as described, when using summary functions, is biased towards adjusting long distance characteristics over short distance local characteristics. Furthermore, this basic algorithm does not account for preference of certain summary characteristics over another, nor can it incorporate non-parametric constraints beyond hardcore inhibition.

If the purpose of an analysis was only to reconstruct a point pattern by only matching short-range local summary characteristics, one can simply define the summary functions $f_j(r)$ over a smaller range $r \in [0, R'_j]$ where $R'_j < R_j$ represents the maximum radius. However, doing so would completely ignore long-range characteristic. Instead, we propose a new energy function for summary functions:

$$E_{f_j}(\psi) = \int_0^{R_j} \frac{[\tilde{f}_j(r;\varphi) - \hat{f}_j(r;\psi)]^2}{\sigma^2(f_j, r)} \mathrm{d}r \text{ for } j = 1, \ldots, J$$

where $\sigma^2(f_j; r)$ is the variance of the summary function $f_j(r)$ on a homogenous Poisson process. This weight serves to balance the distance between the two curves across the

entire range $[0, R_j]$ as lower values of r naturally have smaller squared distances.

In the case of the K-statistic, this variance has been approximated analytically by Ripley [34] in the planar case as:

$$\sigma^2(K_j, r) = \frac{2}{\lambda^2}\left(\frac{\pi r^2}{A(W)} + 0.96\frac{U(W)}{A(W)^2} + 0.13\lambda\frac{U(W)}{A(W)^2}r^5\right)$$

where $U(W)$ is the perimeter of the spatial window W.

Doing so produces reconstructed point patterns that look very similar to the previous algorithm (Figure 3.12) but local characteristics match closer as can be seen by comparing simulation envelopes of the balanced and non-balanced reconstruction algorithm in Figure 3.13 for small values of r. Here, simulations were run for an equal number of iterations to demonstrate that the balanced algorithm does not favour improvements at higher search radii.

Furthermore, to allow for preference of matching one characteristic (either summary statistic or summary function) over another, we propose a new total energy function:

$$E(\psi) = \left(\sum_{i=1}^{I}\alpha_i E_{n_i}(\psi) + \sum_{j=1}^{J}\beta_j E_{f_j}(\psi)\right).$$

where α_i and β_j are pre-defined weights for summary characteristics n_i and $f_j(r)$ respectively. Non-parametric constraints can also be added as an indicator function

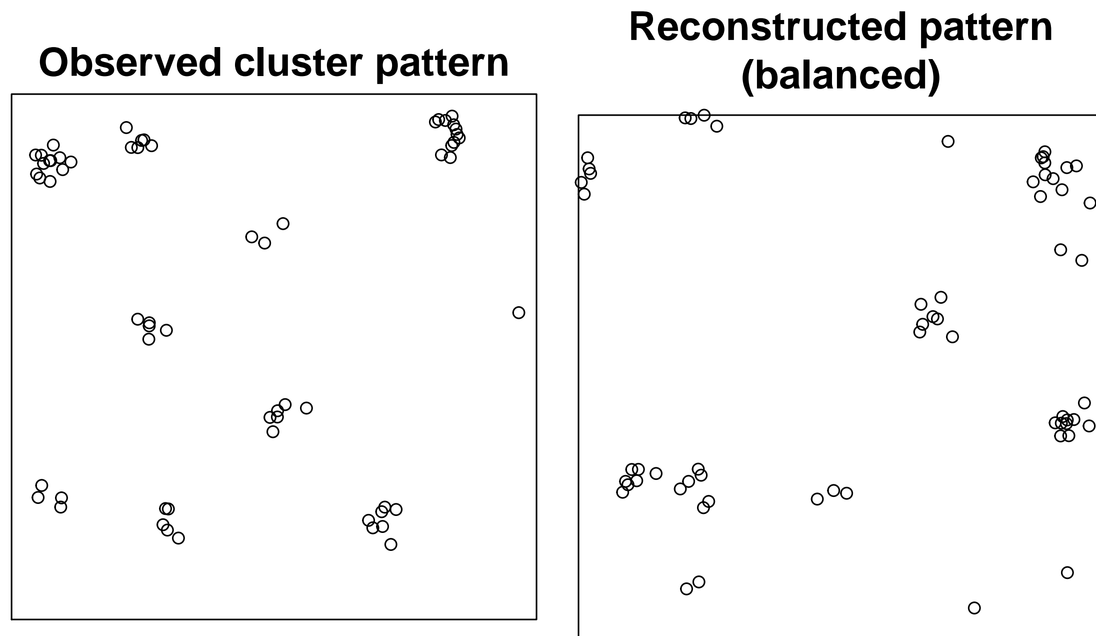

Figure 3.12: The observed point pattern (left) and a balanced reconstructed point pattern (right).

to the equation.

3.5.3 Parallelizing reconstruction

There are several obvious ways that such an algorithm can be parallelized. The first is to parallelize the main loop and selecting the reconstruction with minimum energy. Such an algorithm will result in throwing away $p-1$ proposed moves. A

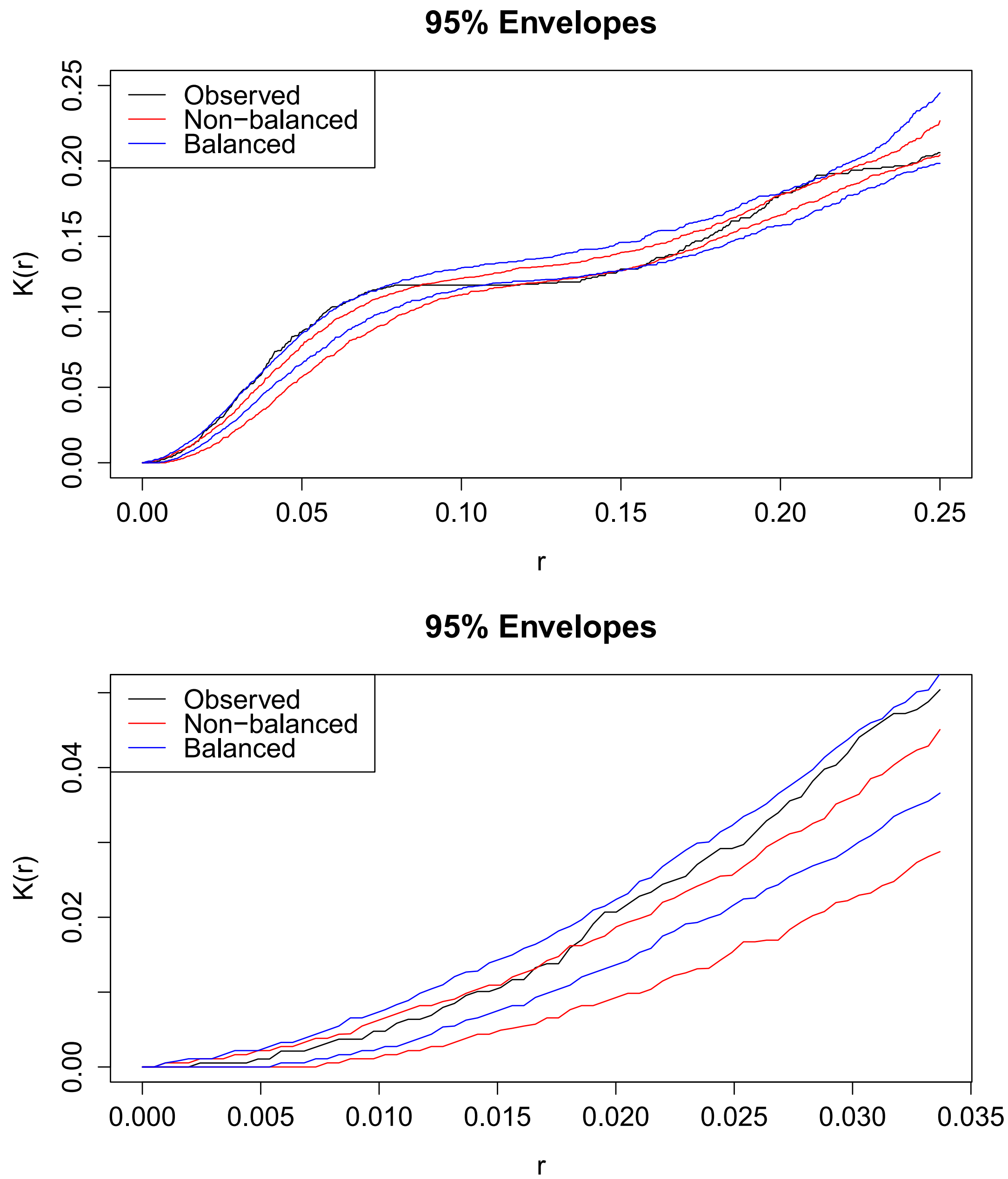

Figure 3.13: Comparison of simulation envelopes of K-function for the observed point pattern, reconstructed point pattern, and a balanced reconstructed point pattern. Bottom graph shows the effect of a balanced algorithm in better matching at small values of r. The top and bottom lines of each colour indicate the upper and lower bounds of each envelope.

simple improvement would be to ensure no two processors will attempt to move the same point to fully explore all moves as quickly as possible. A more complicated improvement involves combining all moves that resulted in a reduced energy, however, this may not be optimal if two proposed moves have a negative interaction.

An algorithm for reconstruction using $c + 1$ processors, $p_0, \ldots, p_c$, working in parallel where p_0 is the manager and $p_1, \ldots, p_c$ are the workers.

Parallel reconstruction algorithm

Given: An observed point pattern φ in W_{obs}, summary characteristics $\tilde{n}_i(\varphi)$ and $\tilde{f}_j(r; \varphi)$, energy change threshold $\varepsilon > 0$, maximum total iterations S, a maximum of iterations without significant change m, .

Step 1. Generate a random point pattern ψ_1 on p_0 from a binomial process with n points where $n = \lfloor \lambda * A(W) + 0.5 \rfloor$.

Step 2. Set $s = 1$

Step 3. do while $E(\psi_{s-m}) - E(\psi_s) < \varepsilon$ and $s < S$

(a) Scatter ψ_s to $p_1, \ldots, p_c$

(b) On each worker p_i in $p_1, \ldots, p_c$:

i. Create $\psi'_{s,i}$ by perturbing a random point in ψ_s.

ii. Send $E(\psi'_{s,i})$ to manager p_0.

(c) Update ψ_{s+1}

$$\psi_{s+1} = \psi'_{s,i} \text{ where } i = \arg\min_i E(\psi'_{s,i})$$

(d) Set $s = s + 1$.

Step 4. ψ_s is a reconstruction of φ.

A simulation was carried out in which a Matern cluster process with $C \sim Poisson(12)$ homogeneous cluster centres, cluster radius 0.04, and $\mu = 7$ mean points per cluster was generated in a unit spatial window. Using this pattern as a target for reconstruction with the K-function as the target function, the parallel reconstruction algorithm was run with $P = \{1, 2, 4, 8, 32\}$ processors to observe the decrease in energy function for varying number of processors. The results of the simulation are seen in Figure 3.14. It can be seen that using more processors in parallel to discover better moves allows the energy function to decrease more quickly. However there are obvious diminishing returns as using 32 processors does not offer substantial improvements over using just 8 processors.

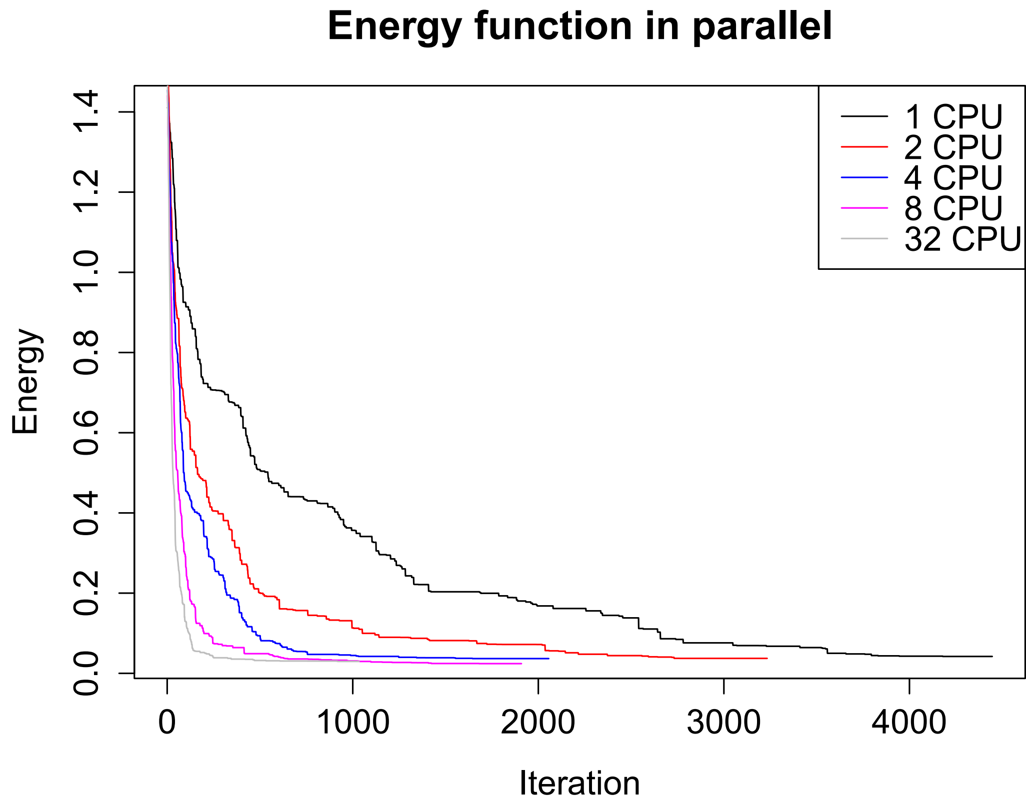

Figure 3.14: Energy function decrease for the reconstruction algorithm parallelized on various number of processors.

3.6 The *parspatstat* package

A package for the R statistical computing language [31] called **parspatstat**, has been implemented to extend some of the functions in **spatstat** [3] in a parallel setting by dividing the estimation into smaller portions while still maintaining proper edge correction where appropriate. It is built on top of **Rmpi** [50], the R implementation of the message passing interface (MPI) framework. In addition to computing traditional statistical summary functions in parallel, it can also speed up simulation of envelopes for model checking and comparison. There is also support for the reading of large datasets in parallel to divide up a spatial point pattern into smaller patterns that each worker can locally store and work on.

The **Rmpi** and **spatstat** packages are required to be set up and installed on all computers that are to be included in the parallel environment (see respective documentation for instructions). First, one must either spawn the desired number of workers (slaves) or launch through a batch job scheduler with the desired number of workers before invoking any of the parallel functions in **parspatstat**. Any data passed into the parallel functions of **parspatstat** are automatically propagated to all workers but any other packages or dependencies that the workers require will need to be loaded explicitly.

```
library(Rmpi)

mpi.spawn.Rslaves(nslaves=8) #spawn 8 workers

mpi.bcast.cmd(require(spatstat)) #have workers load spatstat library
```

3.6.1 Function usage

Names for parallel functions in **parspatstat** are the same as functions in **spatstat** but with a **par** preceding it. For example, the K and L functions, **Kest** and **Lest**, are **parKest** and **parLest**. For all functions, arguments are defined in the same order and passed directly to their **spatstat** counterparts. A few additional parameters specific to parallelization are required as well:

job.num=2*(mpi.comm.size(comm)-1)

Functions in **parspatstat** attempt to make use of all available workers on a specified communicator. Load balancing is implemented by specifying the number of jobs to split the work into through the **job.num** argument. If this value is larger than the number of available workers, then excess jobs are sent to workers as they become available. The default job number is twice the number of available workers and works fairly well if each job is of a comparable complexity. If the complexity varies quite a bit, one may wish to split the work into more jobs since the package can assign jobs to the workers in decreasing order of complexity to balance the overall

load across all workers. However, the more jobs there are, the more overhead that is required for communication. In the case of the summary statistics like the K-function, the **parKest** function estimates complexity of a job by the number of points in a particular sub-window.

comm=1

The communicator on which the workers have been spawned – all available workers from the communicator will be used. If one wishes to use only a subset of the workers, workers may be attached to a separate communicator and that number should be set as the argument instead.

verbose=FALSE

This controls if output of each step should be printed. It can be very useful in debugging if the function appears to have stopped as communication between workers is implemented as blocking calls and send/receives need to be matched up exactly.

load.sort=TRUE

Whether or not to sort jobs by estimated difficulty. This will give an improvement when load balancing is used, that is, when the number of jobs is greater than the number of workers.

3.6.2 Datasets that do not fit in memory

Occasionally we may encounter a dataset that is too large to fit into memory itself and hence needs to be read in sequentially in chunks. As an illustrating example, the lightning data set described in section 3.7 is over 400 megabytes and if we were working on a system with less than 400MB of RAM allocated to the R process, we can get workers to maintain portions of the dataset in chunks by passing a **parppp** object created using **parread.ppp** to **parKest** as an argument. For example, if the 400MB lightning datafile was called **lightning.csv** with a header indicating the x and y coordinates as the 5th and 6th column, the following would compute the K function estimate of the entire dataset across all years without ever having the entire dataset exist on a single machine.

```
ltg <- parread.ppp(file="lightning.csv", xy=c(5,6), chunksize=1000,
header=TRUE, localname="ltgppp")
K.all <- parKest(X=ltg)
```

The manager will read in a chuck of data of size **chunksize**, and send the appropriate rows of the data to each worker. The entire spatial window (user supplied or taken as the smallest bounding box of the data) is divided into strips either horizontally or vertically with the number of strips being equal to the number of workers.

Each strip is also extended by the maximum search radius r to ensure that each worker has all the necessary data points to compute the K-estimate without having to communicate with neighbouring workers.

When chunking in data, load balancing is no longer supported since all workers will be required to have the entire dataset in its entirety between them. As such, chunking is beneficial when the original data cannot fit in memory of a single processor, whether it be a manager or a worker. Chunking also reduces the communication bandwidth required, but not necessarily the communication overhead.

3.7 Example: Ontario lightning data

A dataset of lightning strikes in the province of Ontario, Canada was obtained from the Ontario Ministry of Natural Resources. This data consists of 15.4 million lightning strikes from 1992-2010. In this example, an entirely enclosed square region (between $-84°$ and $-80°$ longitude and between $47°$ and $51°$ latitude) is taken for the purposes of the analysis to not only give an easier to manage spatial window but also to allow us to use minus-sampling edge correction to check the various edge correction methods for accuracy. Figure 3.16 shows original plotted data as well as the section from which our data is extracted.

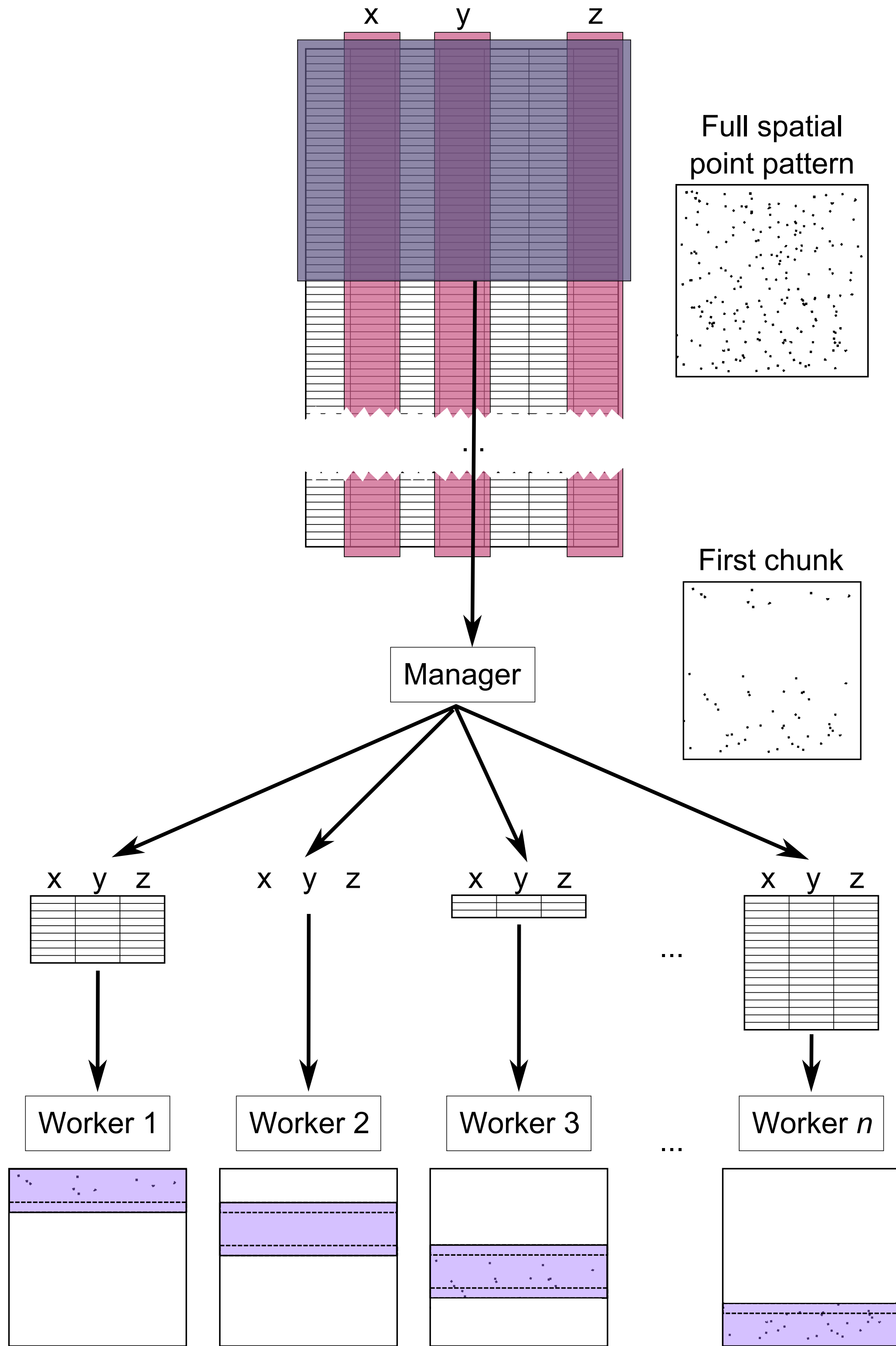

Figure 3.15: Chunking in a dataset and distribution to workers.

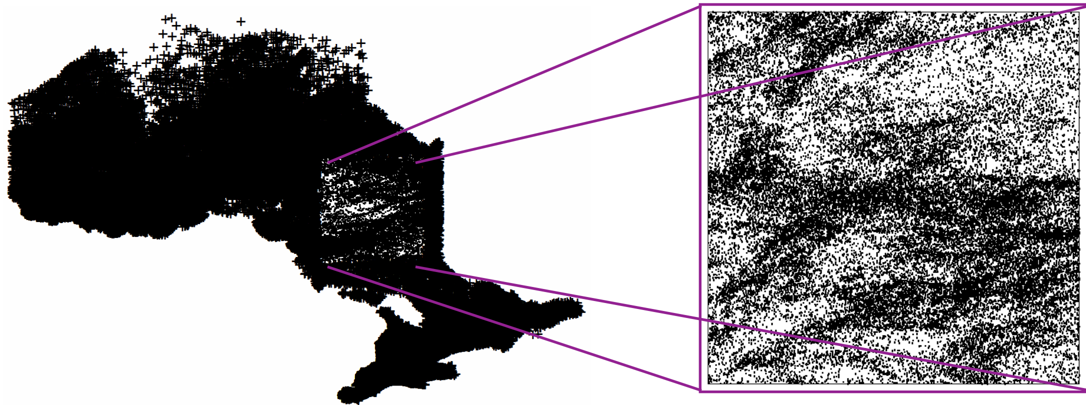

Figure 3.16: Map of all lightning strike data and the extent of the extracted spatial window. The extracted spatial window is plotted with points intead of a + symbol and thus appears lighter.

Looking at the distribution of lightning strikes for each year (Figure 3.17), a question we may be interested in is whether or not there is a difference in point pattern behaviour between years with few strikes and years with many strikes. We will look at only two years from our data, 2003 and 2008, to represent a year with fewer number of strikes (295,533 strikes) and a year with many more strikes (1,216,055), respectively. It can be seen that there is a monthly trend as well when plotting the number of lightning observations by month, but we will ignore this and only look at the yearly aggregated data.

The code below loads the required packages, reads in the lightning data, extracts a sub window and creates a point process pattern (`ppp`) object. The two resulting

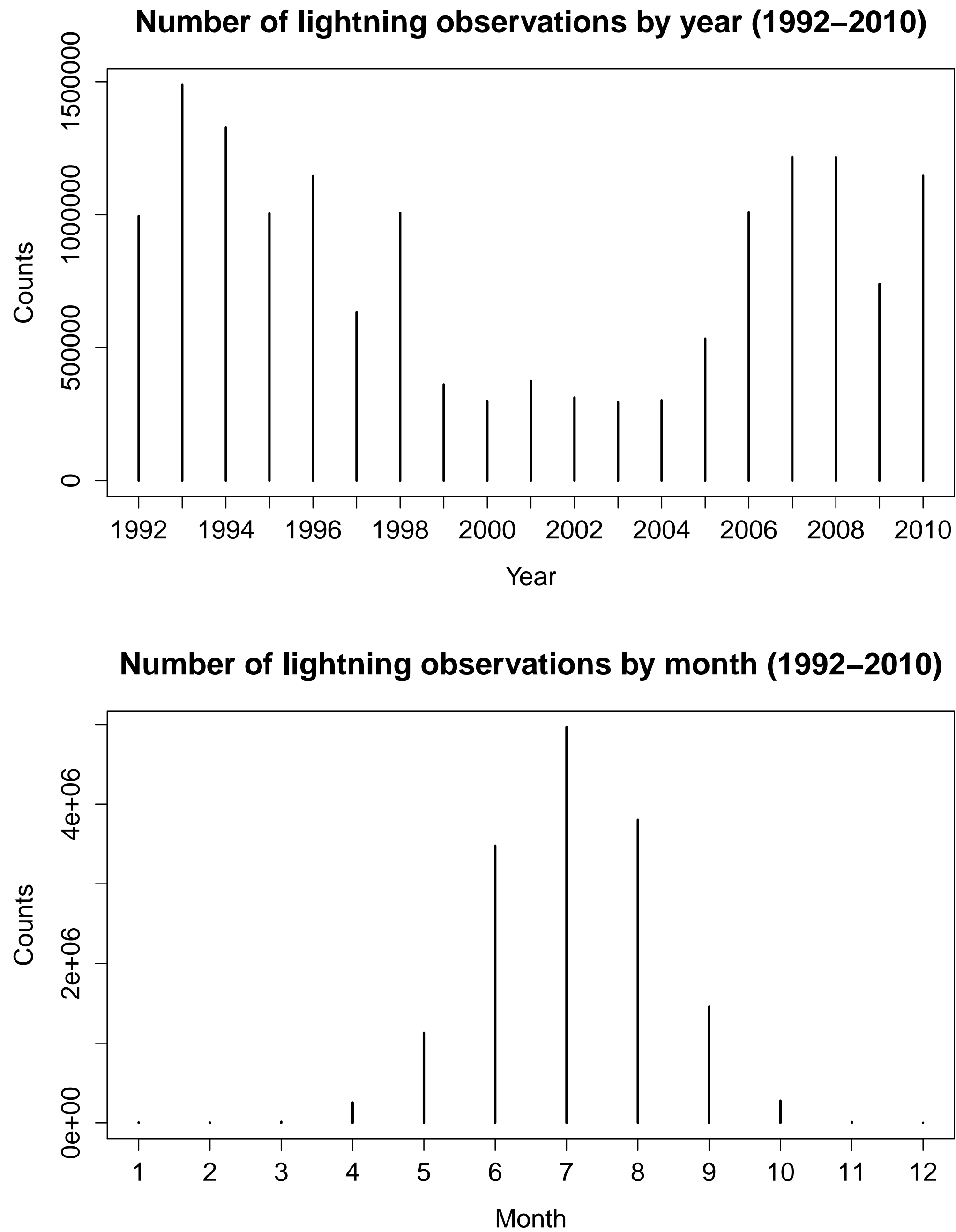

Figure 3.17: Summary of Ontario lightning strike data from 1992 to 2010 by year (top) and by month (bottom).

point patterns are then plotted side by side in Figure 3.18.

```
# Load the required library
library(spatstat)

# Read in the data
data.2003 <- read.csv("lightning/data/ltg2003.csv",header=TRUE)
data.2008 <- read.csv("lightning/data/ltg2008.csv",header=TRUE)

# Define the extent we are interested in
xrange <- c(-84,-80)
yrange <- c(47,51)

# Extract only the rows within our extent
data.2003.sub <- data.2003[ppp.extract(data.2003,owin(xrange,yrange)]

data.2003.sub <- data.2003[which(data.2003$V6 > xrange[1] &
   data.2003$V6 < xrange[2] & data.2003$V5 > yrange[1] &
   data.2003$V5 < yrange[2]),]
data.2008.sub <- data.2008[which(data.2008$V6 > xrange[1] &
   data.2008$V6 < xrange[2] & data.2008$V5 > yrange[1] &
   data.2008$V5 < yrange[2]),]

# Create the spatial observation window
win <- owin(xrange=xrange, yrange=yrange)

# Create ppp objects of the lightning data
ppp.2003 <- ppp(x=data.2003.sub$V6, y=data.2003.sub$V5,window=win)
ppp.2008 <- ppp(x=data.2008.sub$V6, y=data.2008.sub$V5,window=win)

# Plot the patterns of the two years
par(mfrow=c(1,2),mar=rep(0,4))
plot(ppp.2003,pch=".",main="2003")
plot(ppp.2008,pch=".",main="2008")
```

In each of these years, the number of data points in the defined region is too

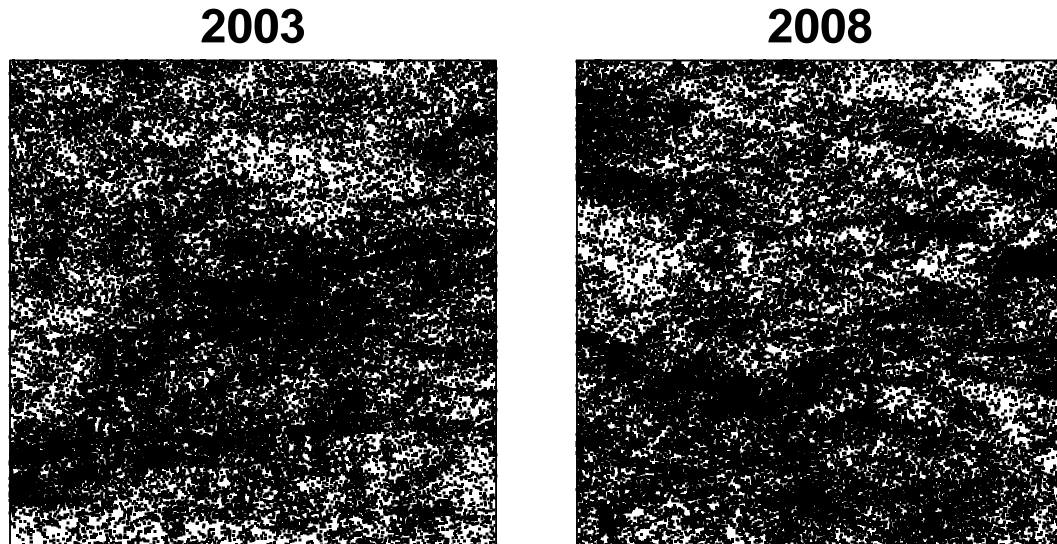

Figure 3.18: Plot of lightning strikes in a region of northern Ontario between 47° and 51° latitude and −84° and −80° longitude in 2003 (left) and 2008 (right).

large (over 60,000 points) for the `Kest` function to compute efficiently if we wish to find edge corrected estimates. The following code snippet loads the required packages, spawns 32 workers, computes the edge-corrected estimates in parallel using the `parKest` function, and plots the resulting K-estimates (shown in Figure 3.19).

```
# Load the parallel library
library(parspatstat)

# Spawn the workers and initialize them
mpi.spawn.Rslaves(nslaves=32)
```

```
mpi.bcast.cmd(require(parspatstat))

# Compute the K-estimate using all corrections in parallel
K.2003 <- parKest(X=ppp.2003, job.num=32*2)
K.2008 <- parKest(X=ppp.2008, job.num=32*2)

# Plot the resulting edge corrected K-estimates
par(mfrow=c(1,2),mar=c(4,4,4,2))
plotK(K.2003,main="2003 K-estimate")
plotK(K.2008,main="2008 K-estimate")
```

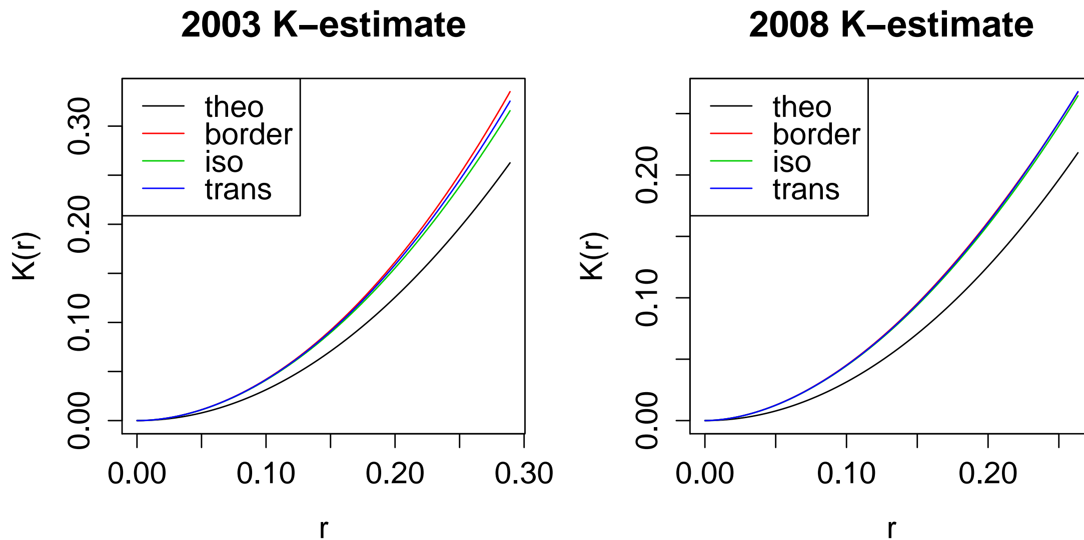

Figure 3.19: Plot of K-estimates of lightning strikes in a square region of Ontario in 2003 (top) and 2008 (bottom) using various border correction methods along with the theoretical line.

If we wish to reconstruct a point pattern that matches a the 2003 lightning data on the K-estimate produced above in a different spatial window, we can do so by specifying a spatial extent and using the `reconstruct` function in **parspatstat**. The

function on which we are matching can be specified to the `parKest` function so it can be computed in parallel. Alternatively, if we were interested in reconstructing only a smaller region that does not require the use of the `parKest` function (but rather can simply use the `Kest`), we can opt to parallelize the reconstruction steps using the `parreconstruct` function. The results of the reconstruction and resulting plots of the K-function are shown in figure 3.20 with resulting energy function less than 0.00001.

```
# Define the spatial extent we are interested in reconstructing
xrange <- c(-81,-80)
yrange <- c(50,51)

# read in data
data.2003 <- read.csv("lightning/data/ltg2003.csv",header=TRUE)
data.2003.full <- data.2003[which(data.2003$V6 > xrange[1] &
   data.2003$V6 < xrange[2] &
   data.2003$V5 > yrange[1] &
   data.2003$V5 < yrange[2]),]

# convert data to ppp
win.full <- owin(xrange=xrange, yrange=yrange)
X.full <- ppp(x=data.2003.full$V6, y=data.2003.full$V5,window=win.full)

# reconstruct a point pattern matching the K estimate
win.small <- owin(xrange=xrange2, yrange=yrange2)
X.small <- X.full[win.small]
Y <- reconstruct(X.small,fun="Kest",eps=0.00001,m=500,maxiter=10000)

# plot the original and reconstructed patterns
par(mfrow=c(1,2))
plot(X.full,pch=".")
plot(Z$ppp,pch=".")
```

```
# compute and plot the resulting K functions
K.full <- parKest(X.full,nlarge=Inf)
K.Y <- Kest(Y$ppp,nlarge=Inf)
par(mfrow=c(1,1))
plot(y=K.full$iso,x=K.small$r,type="l",col=2)
lines(y=K.full$theo,x=K.small$r,col=1)
lines(y=K.Y$iso,x=K.small$r,col=5)
```

Likewise, if we wish to reconstruct a point pattern extended to a larger spatial extent than the original whilst still matching on one or more summary statistics, one can do so with the `reconstruct` or `parreconstruct` method passing in the larger window as an argument and specifying the conditional reconstruction parameter to true. Figure 3.21 shows a reconstructed pattern with the conditional points highlighted.

```
# take a smaller piece of that as our conditional data
xrange2 <- c(-80.5,-80.1)
yrange2 <- c(50.3,50.6)

win.small <- owin(xrange=xrange2, yrange=yrange2)
X.small <- X.full[win.small]

# reconstruct a point pattern in a larger spatial window
Z <- reconstruct(X.small,n=X.full$n,fun="Kest",win=win.full,eps=0.00001,
   m=500,maxiter=10000,conditional=TRUE)

# plot the original and reconstructed patterns
par(mfrow=c(1,2))
plot(X.full,pch=20,main="Original Point Pattern")
plot(X.small,pch=20,add=T,cols=2,main="") #highlight conditional piece

plot(Z$ppp,pch=20,main="Reconstructed Point Pattern")
plot(X.small,pch=20,add=T,cols=2,main="") #highlight conditional piece
```

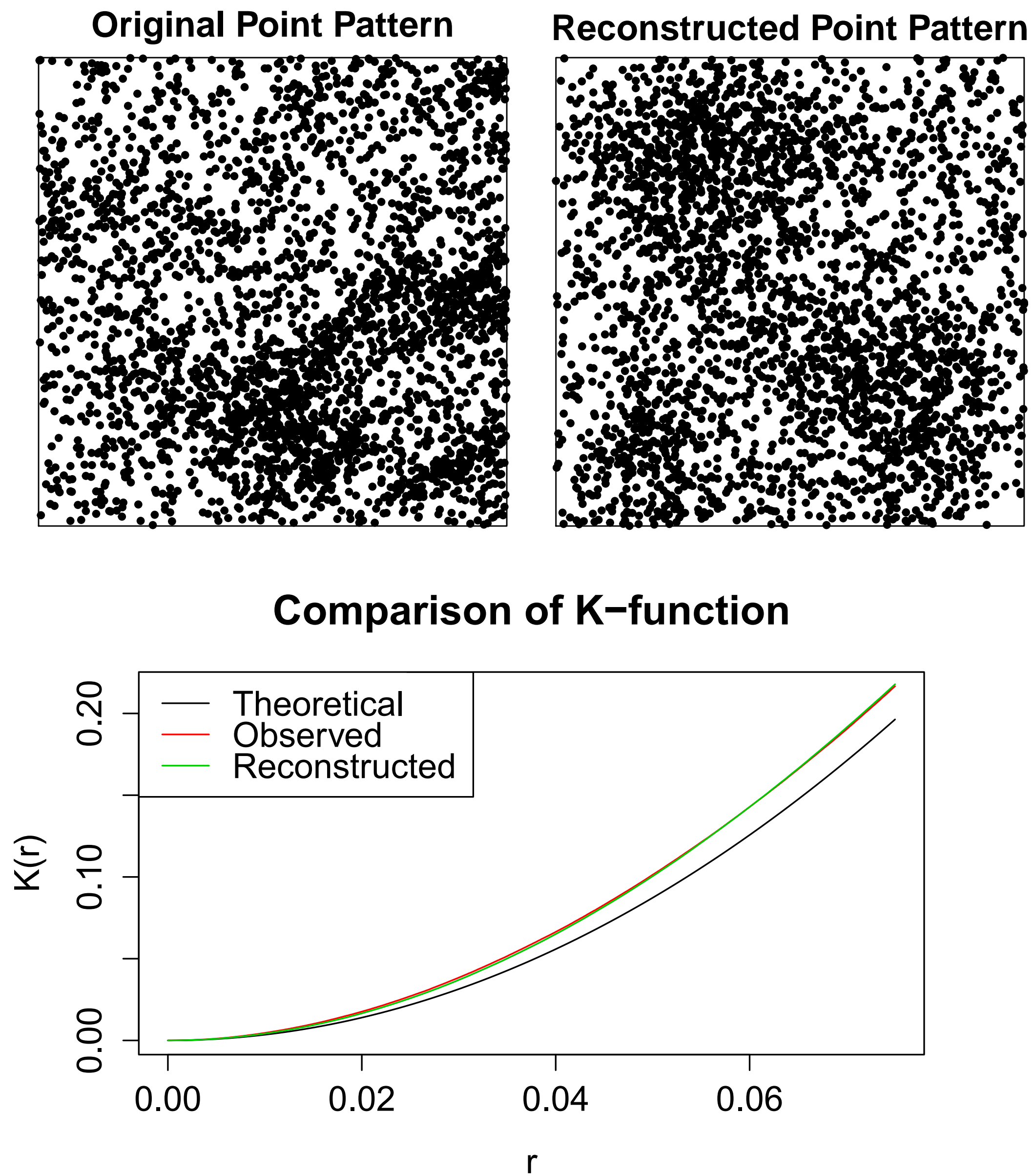

Figure 3.20: The original and reconstructed point patterns (top) and the corresponding K-functions (bottom).

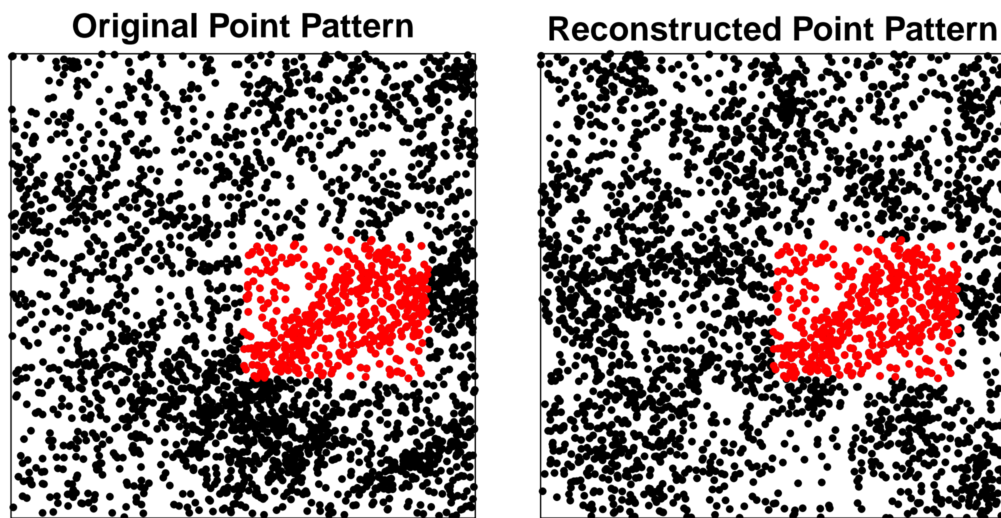

Figure 3.21: The original (left) and reconstruct point patterns (right) conditioning on the points in red.

Chapter 4

Parallel computing for lattice models

4.1 Motivating example: Lattice Fire Spread Model

A stochastic lattice model can be used to simulate fire spread. One such model was described by Boychuk et al. [7] where each pixel of the model can be in one of three states: combustible/unburned fuel, burning fuel, or burnt out/non-fuel. One state can transition only to the next with burnt out/non-fuel acting as an absorbing state. Each pixel may also contain external covariate information such as fuel type, fuel moisture, wind speed, wind direction, and topographic characteristics. Transition from one state to the next is independent and exponentially distributed, making it essentially a continuous-time Markov chain where transition rates and probabilities from one state to another is dependent on the state of a pixel's immediate neighbours

and its external covariate information. A simplified version of this model will be used here instead. The set of neighbours of a point (i, j) are the four surrounding pixels,

$$N(i, j) = \{(i, j), (i, j + 1), (i, j - 1), (i + 1, j)(i - 1, j)\}$$

The set of states in the neighbourhood of (i,j) at time t is given by,

$$X_{N(i,j)}(t) = \{x_{(i,j)}(t) : (i, j) \in N(i, j)\}$$

Figure 4.1 gives a simulated run of a simple fire spread model with uniform wind on a lattice divided on 4 processors.

4.2 Benchmarking and scaling issues

Using an implementation of the above fire growth model, a simulation study was run to create a benchmark of run times of a single problem given varying number of processors. This simulation study was done on a local Beowulf cluster, *karl*, which consists of 56 cores though a maximum of only 24 can be allocated at a time. A square lattice is divided into a number of sub-lattices and two ignition points were randomly distributed in each sub-lattice. Fires are grown from each ignition point following the transition rules described above with a uniform wind field that is directed exactly north east for a fixed number of time steps.

The implementation of this model is very memory intensive as it keeps track of all historic lattices as well. It was seen that memory limitations scaled linearly with

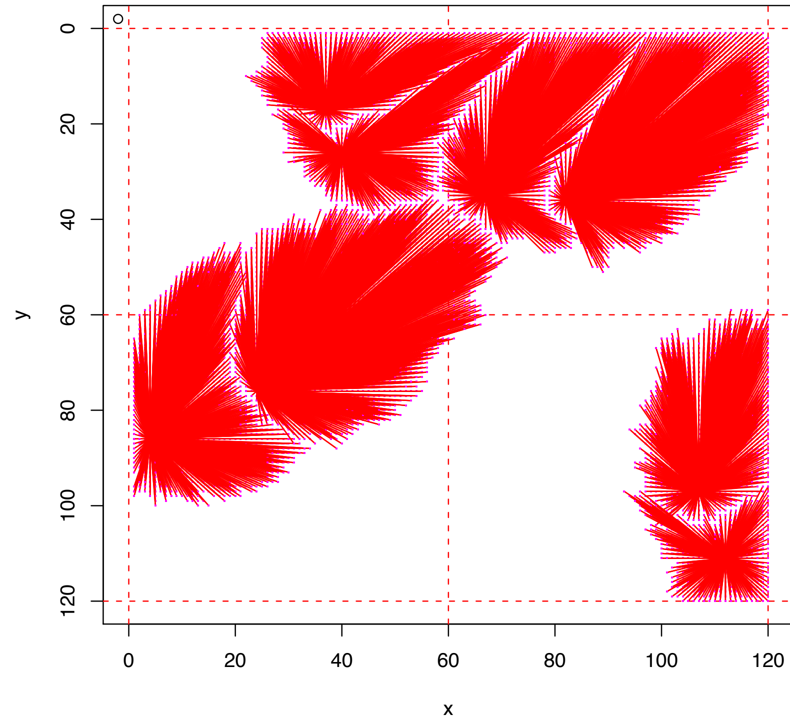

Figure 4.1: Simulation of a simple fire spread model with uniform wind on a 120x120 lattice. Red dotted lines represent division of sub-lattices to 4 CPUs.

the number of processors. That is, the same problem was able to complete twice as many time steps on four CPUs as it could on a two CPUs before running out of memory. This is dependent on the hardware setup as multiple cores on the same node may share the same physical memory. This demonstrates the usefulness of parallel computing (over traditional computing or even multithreading).

In terms of computation time, it can be seen from the run on a 1200x1200 lattice

(Figure 4.2) that the speed up is not linear. Doubling the number of CPUs from two to four results in a bit more than half the computation time on average. The extra time can be partially attributed to randomness but is mostly due to the communication overhead. This overhead goes up with more processors as communication becomes more numerous and occurs more frequently. Eventually, with too many CPUs the communication overhead will overshadow any gained benefits from parallelizing. An extreme case would be to imagine every pixel managed by a different processor. One can imagine the amount of communication required between many pairs of processors in order to compute the value for one pixel.

On a smaller 240x240 lattice (Figure 4.3), communication overhead overshadows gained benefits much sooner. A drop in computation time is also witnessed between 8 and 12 CPUs. This can be attributed to the change in topographic structure as 10 CPU is divided into 5x2 sub-lattices while 12 CPUs is divided into a more geometrically optimal 4x3 structure as described in section 2.2.2. Had a 6x2 division of 12 CPUs been used, this decrease in computation time would not be expected, demonstrating the importance of using an appropriate lattice division scheme. However for computationally intensive enough programs, the communication time becomes negligible compared to the computation time. Hence, fast models do not gain much benefit, and in fact could be slower due to the communication between CPUs acting as a bottleneck.

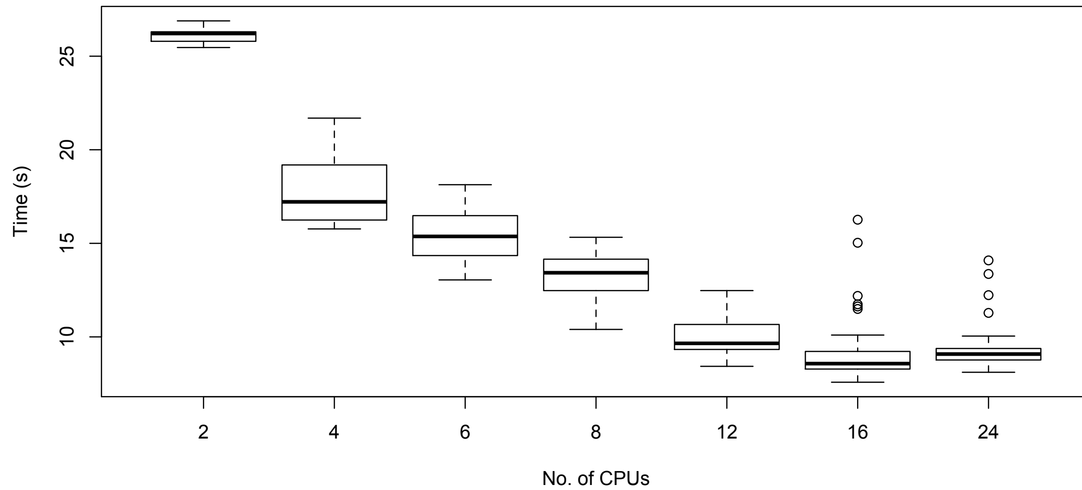

Figure 4.2: Computation time of 48 fires distributed amongst 2 to 24 CPUs on a 1,200 by 1,200 pixel lattice.

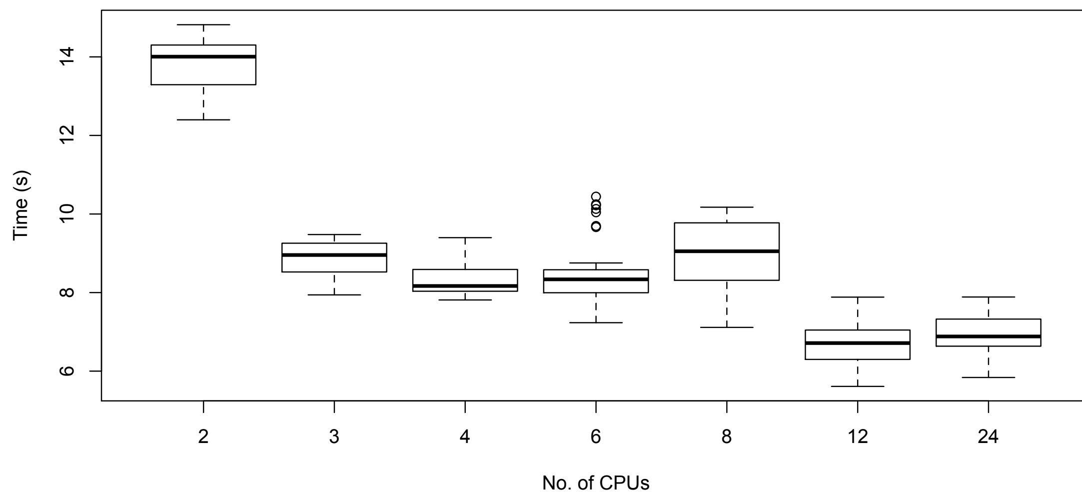

Figure 4.3: Computation time of 48 fires distributed amongst 2 to 24 CPUs on a 240 by 240 pixel lattice.

4.3 Optimizations

4.3.1 Time barrier

Individual processors will finish at different times due to factors such as CPU speed and the complexity of a problem assigned to a particular processor. For locally dependent problems, often processors do not require communication at every time step and in fact may create a large overhead. Instead, we allow each processor to run independently until it hits a time barrier (the length of which is application dependent). Once all processors have hit a barrier or have reached a boundary, we then see if synchronization is required and if so then rollback all processors to the earliest time step in which a barrier or boundary was hit.

An illustrating example is given of three processors going through the exact same steps, with a goal of having each processor reach the 10th time step. Processor 3 is faster than processor 2 which is faster than processor 1. The only difference between the two illustrations is that the second example (Figure 4.4) employs a time barrier of 4, resulting in 6 fewer communication steps while the first example does not employ a time barrier (Figure 4.5).

The idea of a time barrier is most effective when the problem is largely locally dependent and we do not expect crossing the boundary often. In growth models, the time barrier can be useful in the beginning while points are not numerous but as time goes on and activity crosses boundary frequently, any time barrier greater than one would be redundant. Having a time barrier of 1 is essentially the same as communicating at every single time step. For examples such as the heat diffusion

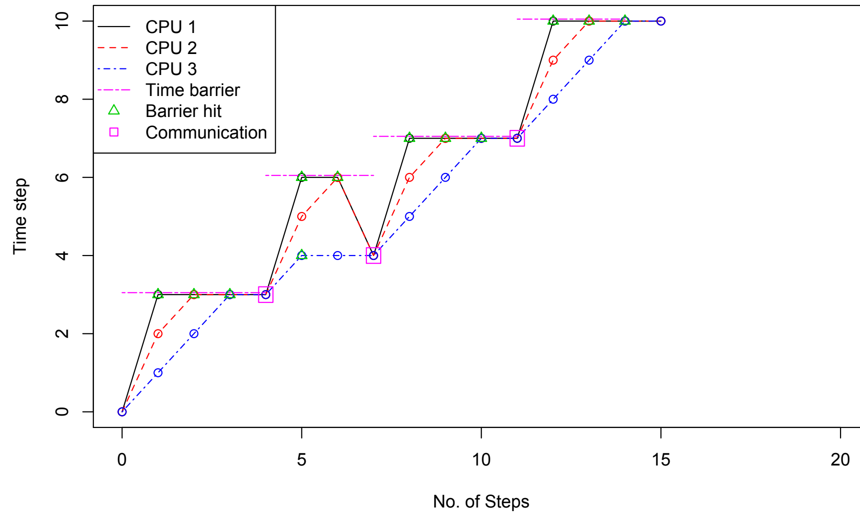

Figure 4.4: Illustration of 3 CPUs working in parallel with communication only when all three CPUs have hit a time barrier or requires swapping.

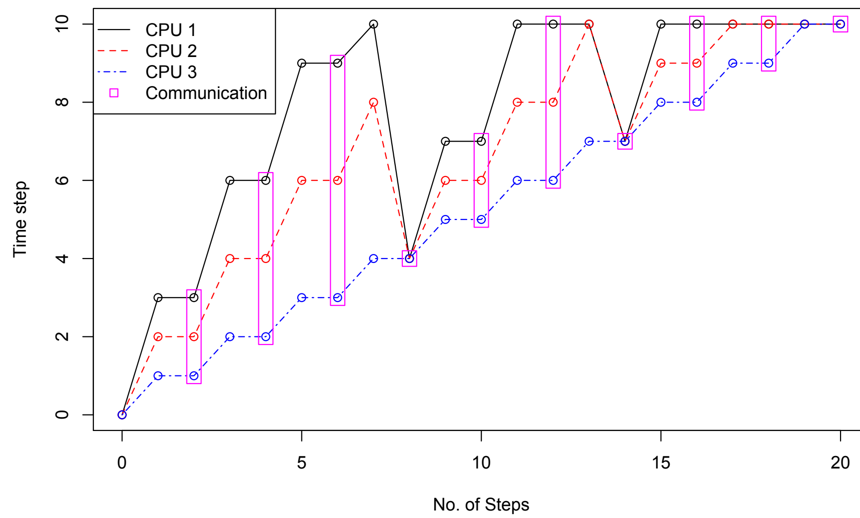

Figure 4.5: Illustration of 3 CPUs working in parallel with communication at every time step to synchronize.

system (Section 2.1.4), there is no benefit to using a time barrier as communication is required at every time step.

4.3.2 Irregular division of lattice

In the forest fire front growth model, a large factor that drives the growth of fire in a particular direction is the wind, both in its speed and direction. If simulating a large landscape with several fires and a prevailing east-west wind, it may be wise to divide the lattice in such a way to minimize the chances of boundary crossings. One possible division would be to divide the lattice into horizontal strips such that ignition points are vertically centered in the division. This division may not be geometrically optimal but will maximize the time to boundary, thereby increasing any benefits offered by using a time barrier and also prolonging its use in the simulation until boundary interaction occurs frequently enough to outweigh gained efficiencies.

Also, on non-homogenous computing systems, one can assign larger or more complex sub-lattices to faster processors in order to more evenly balance the computation time. In the forest fire growth model, a simple measure of complexity can be the number of burning points in a given sub-lattice.

4.3.3 Example: Interacting Particle System

A motivating example of a lattice model used in the next section is an interacting particle system where particles can exist and travel on a lattice according to transition rules. The transition rules of a particular particle are determined by the individual

particle characteristics as well as the characteristics of neighbouring particles. For example, certain particles may repel neighbouring particles whilst others may attract them. Such an example requires a very large lattice in order to accurate represent the amount of detailed required.

4.3.4 Boundary buffer zones

In a stochastic particle interaction system, when particles reach a boundary, there is a higher probability that the particle will cross the boundary again due to its proximity. To reduce the constant communication that may arise in this scenario, a buffer zone is introduced.

Two neighbouring sub-lattices can have an overlapping buffer zone so both sub-lattices will keep track of any activity within the area (Figure 4.6). Upon entering the buffer zone, a particle continues to be simulated by the current processor until it crosses out of the buffer zone and into an area exclusively monitored by the neighbouring processor. Information swapping then occurs at this point and simulation is taken over by the neighbouring processor (Figure 4.7). The advantage of this is that the particle is now within the interior of the neighbouring sub-lattice and hence reduces the probability of it immediately crossing the boundary again.

Sub-lattices can only have horizontal or vertical buffer zones at a given time in order to prevent corner buffer zones that are monitored by more than 2 processors. When creating these buffer zones, the size of the buffer zone will need to be determined as well. Too large of a zone will waste computation resources as that area is

redundantly monitored by two different CPUs while too small of a buffer zone will reduce the benefit gained and may require regular information swapping between processors. This parameter can be determined by such factors as the average distance and direction travelled of particles.

Additionally, interactions within the buffer zone from particles that exist in different processors will need to be kept track of as well. As the simulation becomes more complex and this interaction becomes more probable, one can forgo buffer zones altogether as their cost may begin to outweigh any benefit. In the heat diffusion example (Section 2.1.4), there is no benefit gained from having a buffer zone as every point needs to be computed at every time step. Hence, the use and benefit gained from boundary buffer zones is application dependent.

4.4 Parallel computing for Markov Chain Monte Carlo

In situations where we wish to sample from an intractable distribution, one can use Markov Chain Monte Carlo (MCMC) methods as a common way to do so. For example, in Bayesian inference, we can use MCMC methods to sample from a complex posterior distribution. The Metropolis-Hastings algorithm is a simple way to sample such a chain that generates a candidate for the next sample from a proposal density and accepts or rejects it according to an acceptance ratio. A Metropolis-Hastings algorithm can also be use

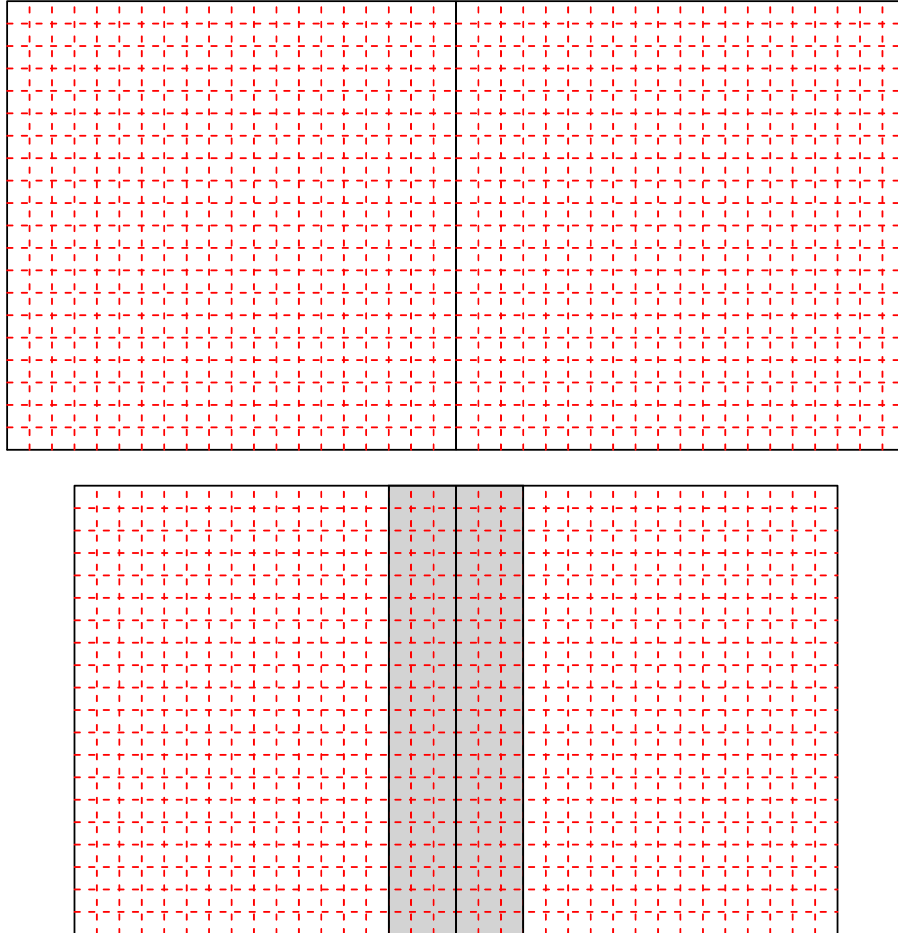

Figure 4.6: Buffer zone to alleviate communication due to boundary crossing, but covers less total area.

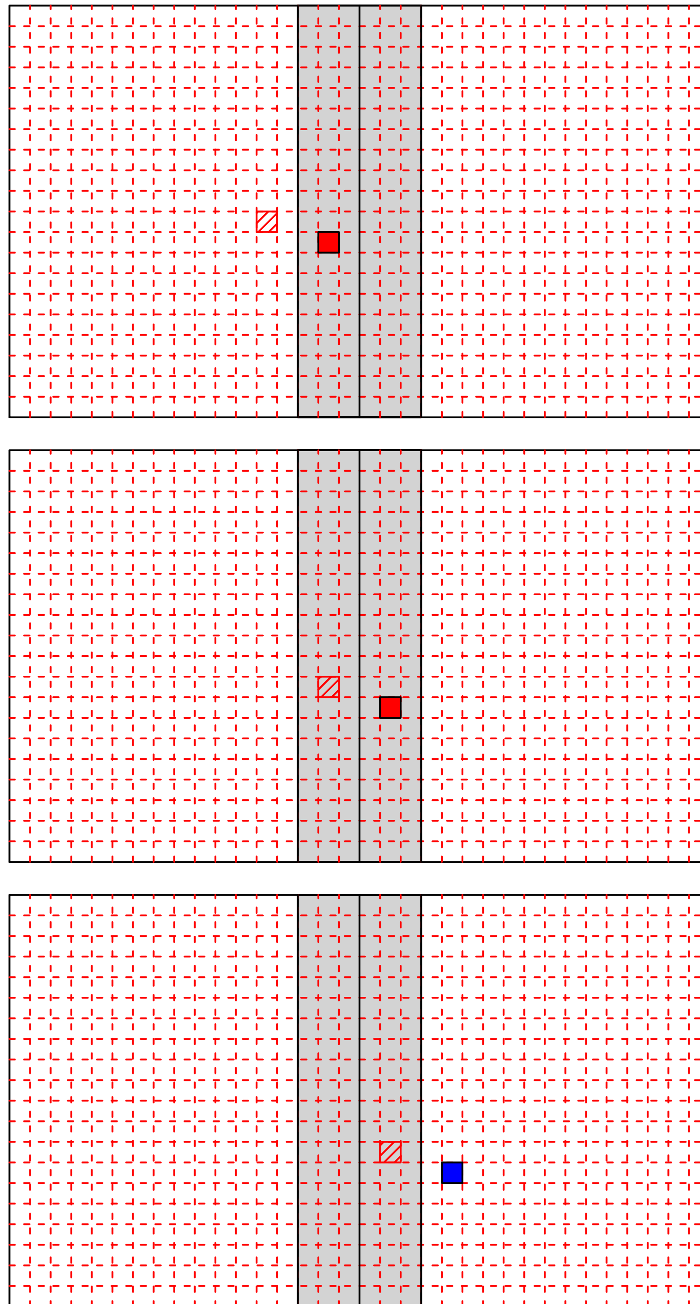

Figure 4.7: Movement of a point (striped) from one processor to another location (solid). The second processor (blue) does not actually receive the point until it passes the buffer zone.

Given some starting value, the chain will converge to realizations of the proper posterior distribution. In practice, one guesses at the amount of time it takes for a chain to reach this equilibrium distribution. This is known as the burn-in time. All samples up until the burn-in time are discarded and the remainder of the chain is taken as draws from the equilibrium distribution. Thinning is also sometimes done, that is, to only keep samples at certain intervals to reduce autocorrelation of draws. The combination of burning and thinning may lead to a lot of computation to be wasted.

With processing power readily available in Beowulf clusters of networked machines working in parallel, there are several ways to use make use of parallel computing in the context of MCMC techniques. In all cases, parallel processors will need to communicate with one another to exchange information (parameter updates) or to simply consolidate results. As before, this is done using a message passing interface (MPI) framework. We assume that each processor runs uniformly at the same speed but Rosenthal [35] provides a treatment for when this is not true. We will make the simplifying assumption that our processors are of uniform speed, which is often the case on Beowulf clusters.

4.4.1 Multiple chain MCMC

Taking advantage of parallel computing in MCMC is not a new concept and is already commonly used in practice. The most direct approach is to generate many Markov chains in parallel, often with a range of starting values. This offers an advantage of

allowing one to see when multiple chains converge which can give an indication of burn-in time. All subsequent draws after this burn-in time from all processors are considered draws from the limiting distribution, assuming that the underlying random number generator is designed to work in parallel so that no processor will not have the same random string as another. One drawback to note about this multiple chain parallelization method is that each individual chain would need to burn-in. A way to mitigate this requirement for burning in is to use a perfect simulation method such as coupling from the past [29] to generate appropriate starting values. Unfortunately it is difficult to implement exact (perfect) sampling for many complex models [48]. In spatial point processes, exact sampling can only be implemented for very simple models.

With a large number of parameters, a Metropolis-Hastings algorithm although easy to implement, may experience poor mixing and long burn-in times. In such a case, the cost of running multiple chains may offset any speed gain. Instead, focus can be placed on parallelizing a single chain.

A Metropolis-Hastings algorithm can be visualized by a Metropolis tree (Figure 4.8) where at each step, a proposal is either rejected (left branch) or accepted (right branch) with some acceptance probability. Each level of the tree represents a single draw from the chain. In the Metropolis tree representations, green cells represent current evaluations and red cells represent realized draws from the chain so far.

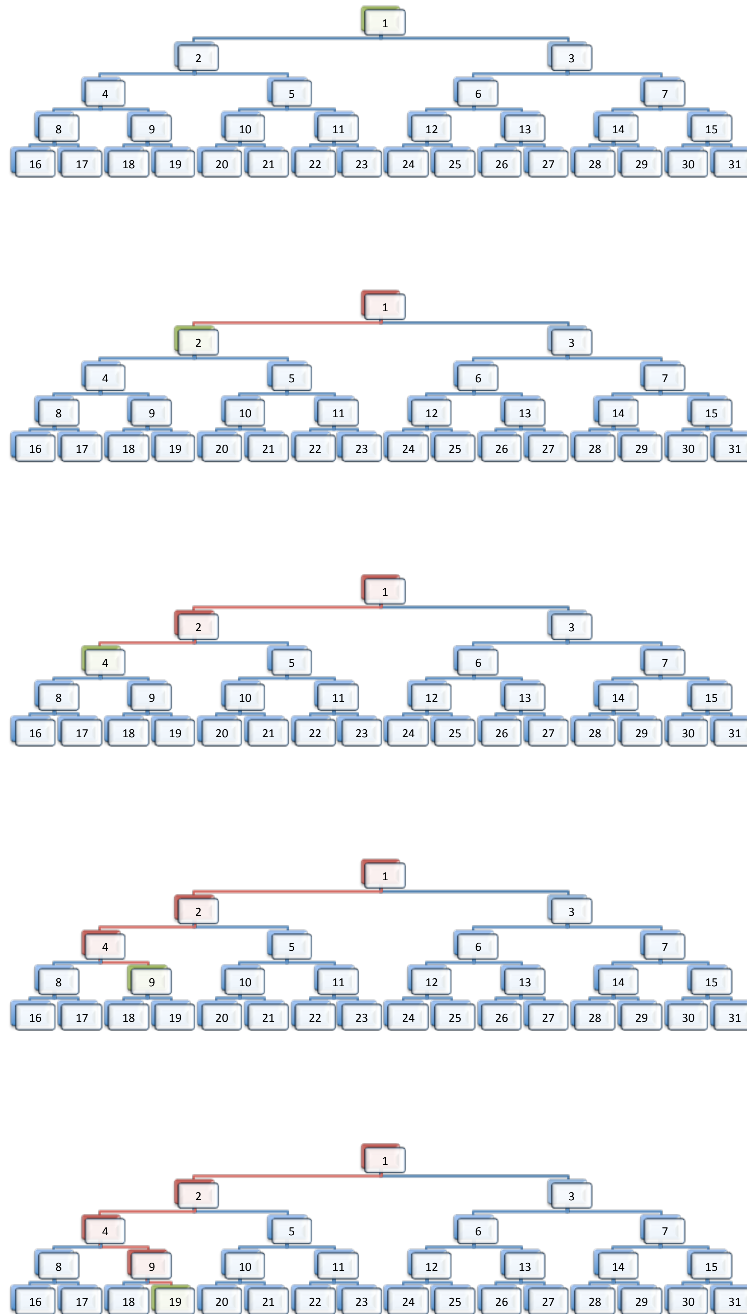

Figure 4.8: Metropolis trees with steps computed serially. At each time step, a single level is completed, representing one draw in the chain. Four draws are computed in four time steps.

4.4.2 Single chain MCMC

One method for parallelizing a single chain involves having each processor start a chain from a predetermined value and generate the chain until that value appears again (regeneration). Individual chains can then be joined together to form a single Markov chain. This method works in discrete state space where the probability of regeneration is non-zero.

A block method of parallelizing Markov chains has also been proposed by Wilkinson [48] for cases where the dependency structure of the underlying model is known and can be divided into blocks. Each of these blocks are conditionally independent given every other block and hence parallel computing can be used to update each block concurrently. This method is limited in the number of processors that can be used, which is the number of blocks the parameters can be divided into, or in the completely independent case, the number of parameters.

Simple pre-fetching

Brockwell [8] introduces an idea of pre-fetching where parallel computing is used to compute all posteriors concurrently at future time-steps. With these posteriors all computed, one can simply jump directly to the appropriate result once acceptance probabilities are calculated. Doing so h steps at a time requires $2^h - 1$ processors (2^h if one includes a manager processor) so this method does not scale well and only offers $log_2(n)$ performance increase where n is the number of processors utilized. An illustration of this pre-fetching method is given in Figure 4.9 where 3 processors

evaluate three nodes in parallel so that two draws can be computed in a single time step. Note that there will be processor evaluations that are thrown out, especially when taking higher numbers of steps ahead.

If the number of processors used does not entirely fill a level, they are allocated left to right from the farthest left branch. Such an allocation scheme is not optimal but typically, one can use exactly 2^h processors.

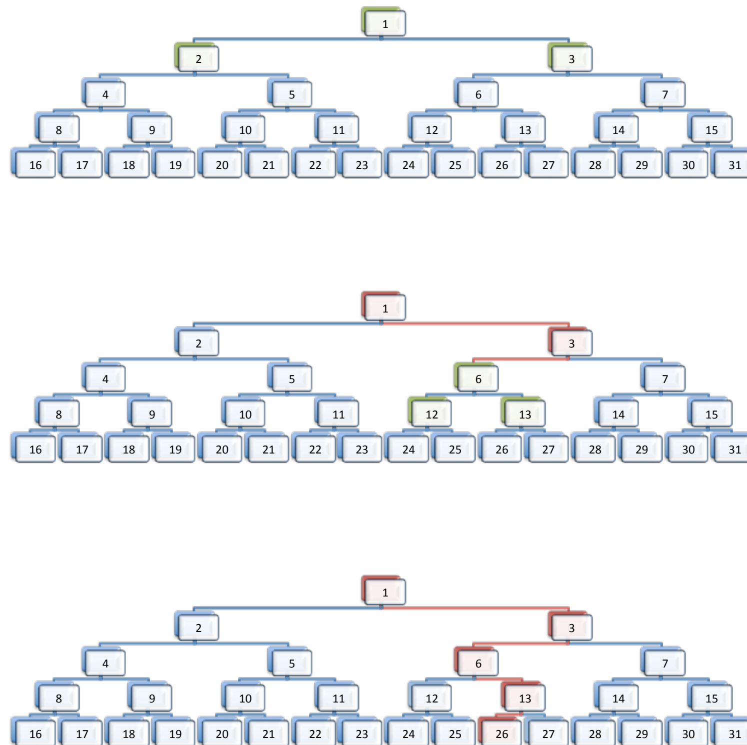

Figure 4.9: Metropolis trees with prefetching two steps at a time using 3 processors. At each time step, two levels are completed, representing two draws in the chain. Four draws are completed in two time steps.

Dynamic pre-fetching

Strid [42] examines several ways to improve on the pre-fetching algorithm by estimating probabilities of acceptance so that only the most likely paths are pre-fetched. Instead of assigning processors to simply evaluate all possibilities in the hth step, some can be assigned further down a path if the probability of taking that path is higher than another outcome on a shallower path. This leads to an improvement over the normal pre-fetching algorithm since more iterations are done on average. Occasionally, the most probable path may not be the actual path and we may end up going fewer steps than the simple prefetching but we expect an greater average number of steps using the same number of processors as simple prefetching. However, this method is still limited in its improvement as ultimately, computation on all but one of the processors will have gone to waste. An example is illustrated in Figure 4.10 where in the first step, the most probable path was in fact wrong at the second draw, but in the second step, the most probable path was correct down three steps.

Through simulation comparing these three methods (Figure 4.11), we can see that the speed up is greatest when using dynamic pre-fetching though in all cases, the amount of speed-up diminishes quickly with the number of processors used. Of interest to note is that there are jumps in improvement when entire levels are filled since it increase the average number of draws per time step, hence the jumps near $2^h - 1$ number of workers.

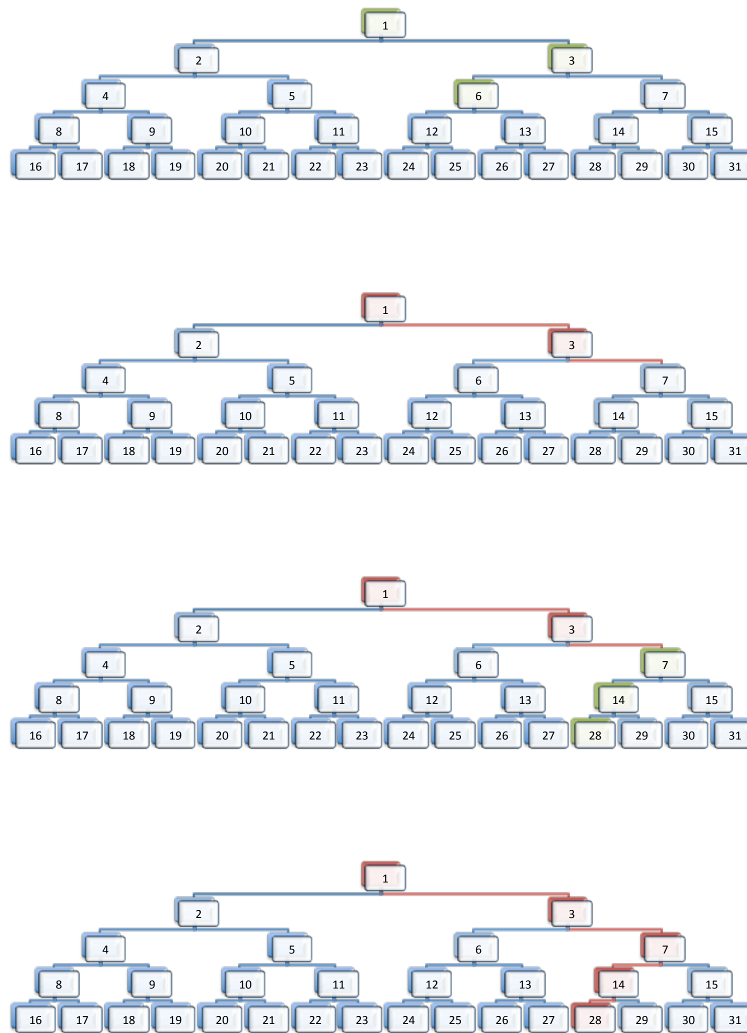

Figure 4.10: Metropolis trees with dynamic prefetching where the most likely paths are evaluated in parallel using 3 processors. At each time step, up to 3 levels are completed. In this particular example, five draws are completed in two time steps.

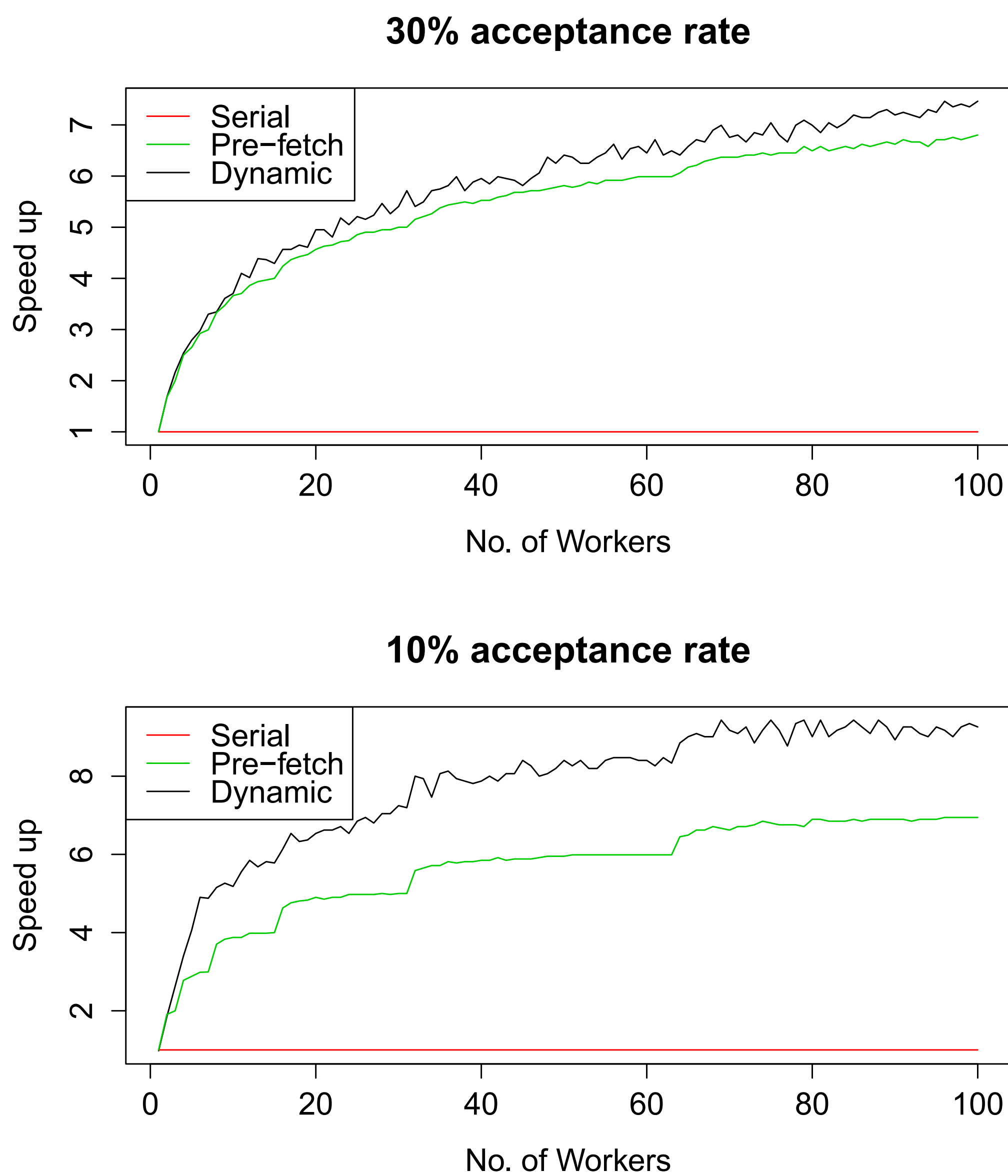

Figure 4.11: Comparison of speed-up versus serial MCMC from using up to 100 processors using simple pre-fetching and dynamic pre-fetching and two different targeted acceptance rates.

4.4.3 Continuous pre-fetching

Each of the previously explored methods for parallelizing a single chain are limited in how many processors they can use before experiencing highly diminishing returns. The idea behind the proposed method is to combine the aforementioned methods to optimally allocate processors in a multi-level parallelization structure. That is, given access to p processors that are assumed to be uniform in performance, what is the optimal distribution of processors as a function of the number of conditionally independent blocks and the approximated acceptance rate at each step?

To illustrate this method, one can imagine a problem where block Metropolis-Hastings can be used to assign two processors to compute the posterior at any given time step. Given only three processors working in parallel, if we are fairly certain of a right branch, then we may be better served using two processors to compute the first node quicker. Figure 4.12 gives an example where there are 7 processors working in parallel. The asterisk represents a node that is computed in parallel in half the time it would take to compute one time step, the first node has already been determined and processors can be allocated further down the line immediately.

Several simplifying assumptions are made. First, it is assumed that evaluating the posterior is significantly more time consuming than the other steps of the algorithm, in fact, the other steps are assumed to be completed in negligible time. Second, we also assume that communication time between processors is also negligible, which is not an unreasonable assumption given specialized hardware and especially when compared with the posterior evaluation time. Finally, we also assume that evaluating

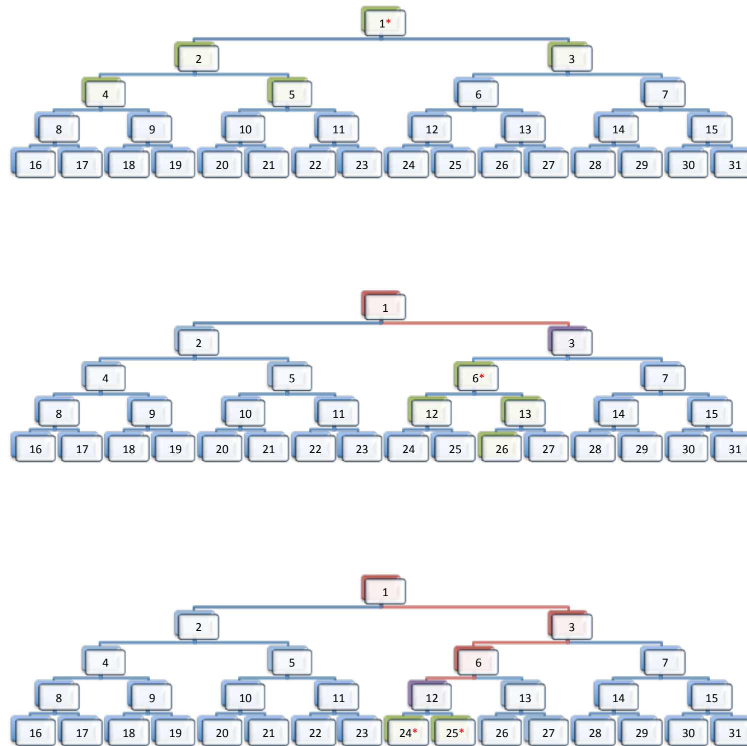

Figure 4.12: Metropolis trees with continuous prefetching where processors can be assigned to higher levels paths to more quickly determine which path is correct and then processors can be reassigned down further down the most probable paths. Asterisks indicate cells that are being computed in parallel and purple cells indicate cells that are already under evaluation.

the posterior is of constant time on each parallel processor, and thus when done sequentially, the time required scales linearly with the number of evaluations.

4.4.4 Comparison with other parallelization techniques

It is recommended in literature that the greatest efficiency gain will come from first using a more efficient algorithm than Metropolis-Hastings but in cases where this is difficult, Metropolis-Hastings serves as a simple and easy-to-implement solution. Next, running chains in parallel will offer near linear scalability, but only if the burn-in time is small compared to the amount of time each processor will spend generating proper samples. It is in the case where the mixing is poor, often in high dimension space, and there is no alternative to a simple Metropolis-Hastings algorithm that parallelization of a single chain MCMC may prove useful.

Chapter 5

Conclusion

In this thesis, several areas of spatial statistics were looked at from a parallel computing perspective to determine methods that can take advantage of parallel computing in a non-trivial way beyond the embarrassingly parallel cases.

Issues that arise when incorporating any sort of parallel programming to spatial data was addressed and an R package was developed that implements many of the functions and methods described.

In spatial point processes, many summary functions can be computed in parallel while still maintaining proper edge correction themes. It was shown that the most useful edge correction methods are also those which require parallelization to handle large datasets. Model evaluation by means of simulation was also parallelized as well on two levels.

An improved point process reconstruction algorithm was proposed and demonstrably performs better than the existing reconstruction algorithm is matching short

range behaviour. A parallel version of the algorithm was also examined with very clear performance benefits. It also adds support for conditional reconstruction of a point pattern to a larger spatial window.

In stochastic lattice models, a framework was created to support simulation of models in parallel while maintaining proper communication between adjacent lattices. Additional ideas for improvement were discussed, though performance improvements are only under specific circumstances.

In Markov Chain Monte Carlo methods, single chain parallelization was examined with different methods to best make use of parallel computing resources. A new multilayered parallelization technique was proposed based on pre-fetching to better utilize parallel computing resources that are available.

5.1 Further Work

The point process methods described work in any number of dimensions though the software implementation only supports two dimensions. Support for three dimensional point patterns can be added in the future to aid in applications such as climate modelling. However, the division of a spatial window in three dimensions is quite a bit more complicated beyond trivial lattice division methods. For example, depending on how a neighbourhood structure is defined, a single voxel may have up to 26 first degree neighbours (if we include ones that touch on a corner) as opposed to a maximum of 8 in two dimensions (Figure 5.1).

The proposed parallel reconstruction algorithm does not make use of all current

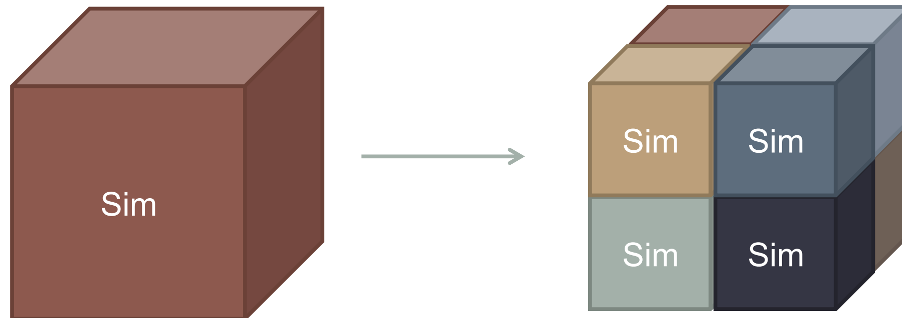

Figure 5.1: Splitting a three dimensional lattice into smaller sub-lattices.

and past information to determine the best move but rather takes the single move that minimizes the energy function at the current step. For example, perhaps some combination of the top three moves computed in parallel can produce a move better than just the top move itself. Other optimization techniques such as a scheme simulated simulated annealing to prevent getting stuck in local minima can be further examined.

The implementation of stochastic lattice models requires the model to be written in a specific way that can fully take advantage of the parallelization. A more general implementation can be examined to allow parallelization of a lattice model that is defined in some structured way so users can easily extend the lattice optimization methods to other applications beyond the simple fire spread model.

Parallelization of single chain MCMC can take advantage of existing MCMC techniques that allow parallel computing at a lower level such as parallel matrix computations to further utilize available resources.

Bibliography

[1] M. P. Armstrong and R. Marciano. Massively parallel processing of spatial statistics. *International journal of geographical information systems*, 9:169–189, 1995.

[2] A. Baddeley, M. Kerscher, K. Schladitz, and B. T. Scott. Estimating the J function without edge correction. *ArXiv Mathematics e-prints*, 1999.

[3] A. Baddeley and R. Turner. Spatstat: an R package for analyzing spatial point patterns. *Journal of Statistical Software*, 12:1–42, 2005.

[4] A. J. Baddeley and B. W. Silverman. A cautionary example on the use of second-order methods for analyzing point patterns. *Biometrics*, 40:1089–1093, 1984.

[5] J. E. Besag. Spatial interaction and the statistical analysis of lattice systems. *Journal of the Royal Statistical Society Series B*, 36:192–236, 1974.

[6] J. E. Besag. Contribution to the discussion of Dr. Ripley's paper. *Journal of the Royal Statistical Society Series A*, 39:193–195, 1977.

[7] D. Boychuk, W. J. Braun, R. J. Kulperger, Z. L. Krougly, and D. A. Stanford. A stochastic forest fire growth model. *Environmental and Ecological Statistics*, 16:133–141, 2009.

[8] A. Brockwell. Parallel markov chain monte carlo simulation by pre-fetching. *Journal of Computational and Graphical Statistics*, 15:246–261, 2006.

[9] M. Cannataro, S. D. Gregorio, R. Rongo, W. Spataro, G. Spezzano, and D. Talia. A parallel cellular automata environment on multicomputers for computational science. *Parallel computing*, 21:803–823, 1995.

[10] N. Cressie. *Statistics for spatial data*. Wiley series in probability and mathematical statistics: Applied probability and statistics. J. Wiley, 1993.

[11] P. J. Diggle. *Statistical analysis of spatial point patterns*. Hodder Education Publishers, 2003.

[12] B. Efron and R J. Tibshirani. *An introduction to the bootstrap*. Chapman & Hall, 1993.

[13] G. François and P. Raphaël. On explicit formulas of edge effect correction for Ripley's K-function. *Journal of Vegetation Science*, 10:433–438, 1999.

[14] A. Getis and J. K. Ord. The analysis of spatial association by use of distance statistics. *Geographical Analysis*, 24:189–206, 1992.

[15] G. H. Givens and J. A. Hoeting. *Computational statistics*. John Wiley and Sons, 2005.

[16] D. A. Griffith. Supercomputing and spatial statistics: a reconnaissance. *Professional geographer*, 42:481–492, 1990.

[17] P. Haase. Spatial pattern analysis in ecology based on ripley's K-function: Introduction and methods of edge correction. *Journal of Vegetation Science*, 6:575–582, 1995.

[18] R. Healey, S. Dowers, B. Gittings, and M. Mineter. *Parallel processing algorithms for GIS*. Taylor & Francis, 1998.

[19] L. P. Ho and S. N. Chiu. Using weight functions in spatial point pattern analysis with application to plant ecology data. *Communications in statistics - Simulation and computation*, 38:269–287, 2009.

[20] J. Illian, A. Penttinen, H. Stoyan, and D. Stoyan. *Spatial analysis and modelling of spatial point patterns*. Wiley and Sons, 2008.

[21] O. Kallenberg. *Foundations of modern probability*. Springer-Verlag, 2002.

[22] M. Kerscher, I. Szapudi, and A. S. Szalay. A comparison of estimators for the two-point correlation function. *Astrophysics Journal Letters*, 535:L13–L16, 2000.

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

[24] V. J. Martinez and E. Saar. *Statistics of the galaxy distribution*. Chapman & Hall, 2002.

[25] J. Osher and D. Stoyan. On the second-order and orientation analysis of planar stationary point processes. *Biometrical Journal*, 23:523–533, 1987. doi: 10.1002/bimj.4710230602.

[26] G. Ostrouchov, W. C. Chen, D. Schmidt, and P. Patel. *Programming with Big Data in R*, 2012.

[27] A. Pommerening and D. Stoyan. Edge-correction needs in estimating indices of spatial forest structure. *Canadian Journal of Forestry Research*, 36:1723–1739, 2006. doi: 10.1139/X06-060.

[28] A. Pommerening and D. Stoyan. Reconstructing spatial tree point patterns from nearest neighbour summary statistics measured in small subwindows. *Canadian Journal of Forestry Research*, 38:1110–1122, 2008. doi: 10.1139/X07-222.

[29] J. G. Propp and D. B. Wilson. Exact sampling with coupled markov chains and applications to statistical mechanics. *Random Structures & Algorithms*, 9:223–252, 1996.

[30] R Development Core Team. *Package 'parallel'*, 2012.

[31] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2012. ISBN 3-900051-07-0.

[32] B. D. Ripley. The second-order analysis of stationary point processes. *Journal of Applied Probability*, 13:255–266, 1976.

[33] B. D. Ripley. *Spatial Statistics*. Wiley series in probability and mathematical statistics: Applied probability and statistics. J. Wiley, 1981.

[34] B. D. Ripley. *Statistical inference for spatial point processes*. Cambridge University Press, 1988.

[35] J. S. Rosenthal. Parallel computing and monte carlo algorithms. *Journal of Theoretical Statistics*, 4:207–236, 2000.

[36] K. Schladitz and A. J. Baddeley. A third order point process characteristic. *Scandinavian journal of statistics*, 27:657–671, 2000.

[37] M. Schmidberger, M. Morgan, D. Eddelbuettel, H. Yu, L. Tierney, and U. Mansmann. State of the art in parallel computing with R. *Journal of Statistical Software*, 31:1–27, 2009.

[38] M. Snethlage. Is bootstrap really helpful in point process statistics? *Metrika*, 49:245–255, 1999.

[39] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI - The Complete Reference Volume 1, The MPI Core*. The MIT Press, 2001.

[40] R. W. Sterner, C. A. Ribic, and G. E. Schatz. Testing for life historical changes in spatial patterns of four tropical tree species. *Journal of Ecology*, 74:621–633, 1986.

[41] D. Stoyan and A. Penttinen. Recent applications of point process models in forestry statistics. *Statistical science*, 15:61–78, 2000.

[42] I. Strid. Efficient parallelization of metropolis-hastings algorithms using a prefetching algorithm. *Computation Statistics & Data Analysis*, 54:2814–2835, 2010.

[43] J. Szwagrzyk and M. Czerwczak. Spatial patterns of trees in natural forests of east-central europe. *Journal of Vegetation Science*, 4:469–476, 1993.

[44] L. Tierney, A. J. Rossini, N. Li, and H. Sevcikova. *multicore: parallel processing of R code on machines with multiple cores or CPUs*, 2003.

[45] L. Tierney, A. J. Rossini, N. Li, and H. Sevcikova. *Simple network of workstations for R*, 2003.

[46] A. Tscheschel and D. Stoyan. Statistical reconstruction of random point patterns. *Computational statistics & data analysis*, 51:859–871, 2006.

[47] M. N. M. van Lieshout and A. J. Baddeley. A nonparametric measure of spatial interaction in point patterns. *Statistica Neerlandica*, 50:344–361, 1996.

[48] D. Wilkinson. *Parallel bayesian computation.* Chapman and Hall, 2006.

[49] I. Yamada and P. A. Rogerson. An empirical comparison of edge effect correction methods applied to K-function analysis. *Geographical Analysis*, 35:97–109, 2010.

[50] H. Yu. Rmpi: parallel statistical computing in R. *R News*, 2:10–14, 2002.

Curriculum Vitae

Name:	Jonathan Lee
Post-Secondary Education and Degrees:	University of Waterloo Waterloo, ON 2003 - 2008 B.Math
	Queen's University Kingston, ON 2006 - 2008 B.Ed.
	The University of Western Ontario London, ON 2008 - 2009 M.Sc.
	The University of Western Ontario London, ON 2009 - 2013 Ph.D.
Honours and Awards:	OGS 2010-2013
Related Work Experience:	Teaching Assistant University of Waterloo 2004 - 2008
	Teaching Assistant The University of Western Ontario 2008 - 2013
	Sessional Lecturer The University of Western Ontario 2011