The Pennsylvania State University
The Graduate School

PARALLEL MULTIVARIATE SLICE SAMPLING

A Thesis in
Statistics
by
Matthew M. Tibbits

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2009

The thesis of Matthew M. Tibbits was reviewed and approved* by the following:

John C. Liechty
Associate Professor of Marketing and Statistics
Thesis Advisor, Chair of Committee

Murali Haran
Assistant Professor of Statistics

Bruce G. Lindsay
Willaman Professor of Statistics and Department Head

*Signatures are on file in the Graduate School.

Abstract

Slice sampling provides an easily implemented method for constructing a Markov chain Monte Carlo (MCMC) algorithm. However, slice sampling has two major drawbacks: (i) it requires repeated evaluation of likelihoods for each update, which can make it impractical when each evaluation is expensive or as the number of evaluations grows (geometrically) with the dimension of the slice sampler, and (ii) since it can be challenging to construct multivariate updates, the updates are typically univariate, often resulting in slow mixing samplers. We propose an approach to multivariate slice sampling that naturally lends itself to a parallel implementation. Our approach takes advantage of recent advances in computer architecture, for instance, the newest generation of graphics cards can execute roughly $30,000$ threads simultaneously. We demonstrate that it is possible to construct a multivariate slice sampler that has good mixing properties and is efficient in terms of computing time. The contributions of this article are therefore twofold. We study approaches for constructing a multivariate slice sampler, and we show how parallel computing can be useful for making MCMC algorithms computationally efficient. We study various implementations of our algorithm in the context of real and simulated data.

Table of Contents

List of Figures

List of Tables

Acknowledgments

The author is very grateful to Dr. Murali Haran and Dr. John Liechty for their guidance and efforts during the course of this research. The author is eternally grateful for the help and support of his wife, Kristina Zeiser. The author is also grateful to the following people for their helpful conversations and suggestions regarding this effort: Chris Groenyke, K. Sham Bhat, Muhammad Atiyat, and Scott Roths.

Dedication

To my Krissy.

Chapter 1

Introduction

It is well known that the Markov chain Monte Carlo (MCMC) algorithm, which is based on the Metropolis-Hastings algorithm, provides a very general approach for approximating integrals (expectations) with respect to a wide range of complicated distributions. When full conditional distributions are non-standard, the process of constructing MCMC algorithms is far from automatic and even when it is possible to sample from the full conditionals via Gibbs updates, the resulting samplers can exhibit poor mixing properties. It is often the case that the Metropolis-Hastings algorithm needs to be carefully tailored to each particular distribution and the search for an appropriate proposal distribution with good mixing properties can be time consuming. The slice sampler (Damien et al., 1999; Mira and Tierney, 2002; Neal, 1997, 2003a) has been proposed as an easily implemented method for constructing an MCMC algorithm and can, in many circumstances, result in samplers with good mixing properties. Slice sampling has been used in many contexts, for example in spatial models: (Agarwal and Gelfand, 2005; Yan et al., 2007), in biological models: (Lewis et al., 2005; Shahbaba and Neal, 2006; Sun et al., 2007), variable selection: (Kinney and Dunson, 2007; Nott and Leonte, 2004), and machine learning: (Andrieu et al., 2003; Kovac, 2005; Mackay, 2002). Slice samplers can adapt to local characteristics of the distribution, which can make them easier to tune than Metropolis-Hastings approaches. Also, by adapting to local characteristics and by making jumps across regions of low probability, a well constructed slice sampler can avoid the slow converging random walk behavior that many standard Metropolis algorithms exhibit Neal (2003a).

A slice sampler exploits the fact that sampling points uniformly from the region under the curve of a density function is identical to drawing samples directly from the distribution. Typically, the slice sampler is used for univariate updates (e.g., sampling from a full-conditional density for a single parameter) or for updates to one variable at a time in a multivariate density. While univariate slice samplers may improve upon standard univariate Metropolis algorithms, univariate samplers in general can mix poorly in multivariate settings, especially when several variables exhibit strong dependencies. In such cases, the mixing of the sampler can be greatly improved by simultaneously updating multiple highly dependent variables at once. But, while multivariate slice samplers have been discussed Neal (2003a), they are rarely used in practice as they can be difficult to construct and computationally expensive to use due to the large number of evaluations of the target distribution required for each update.

In this article, we explore the construction of simple, automatic multivariate slice updates, which take advantage of the latest parallel computing technology available. Since modern computing is moving towards massively parallelized computation rather than simply increasing the power of individual processors, a very interesting and important challenge is to find ways to exploit parallel computing power in the context of inherently sequential algorithms like MCMC. While parallel computing has been explored in a few other MCMC contexts, for instance to accelerate matrix computations when evaluating the target density at a single location Yan et al. (2007), it has yet to be applied to multivariate slice sampling. We consider strategies for constructing efficient multivariate slice samplers that take advantage of parallel computing.

This article makes two primary contributions: (i) we develop multivariate slice samplers and compare their performance to univariate slice samplers, and (ii) we explore how new developments in parallel computing can be used to make computationally expensive multivariate slice samplers fast and practical. The remainder of the paper is organized as follows. Section 2 outlines the univariate and multivariate slice sampling algorithms, Section 3 examines two different software approaches to parallelism (OpenMP and CUDA), Section 4 examines the performance of the various parallel sampling algorithms in the context of both simulated and real data examples invloving a popular class of Gaussian process models, and Section 5 sum-

marizes the effectiveness of the different algorithms and points to future avenues for research.

Chapter 2

Slice Sampling

In this section we provide a brief overview of the basic univariate slice sampler. We then review some of the univariate slice sampler's drawbacks; these motivate the exploration of multivariate slice samplers. We describe multivariate slice sampling and explain how exploiting modern parallel processing can greatly reduce the computational costs of the multivariate slice sampler, resulting in a fast mixing Markov chain which is also computationally efficient.

2.1 Univariate Slice Sampling

The univariate slice sampler outlined here and the multivariate slice sampler outlined in Section 2.2 use the same basic algorithm. Both augment the parameter space with an auxilliary variable and then appropriately construct a region or "slice" from which one can sample uniformly. However, the univariate version augments each parameter with its own auxilliary variable. Hence, it will sample variable-at-a-time using Algorithm 1. For simplicity, we outline the algorithm in the context of a one dimensional parameter β. We then examine its performance in a simple two dimensional regression example. Note that in this section and those that follow we will use subscripts to denote the components of a vector and superscripts to index iterations of the algorithm.

The ith update of β is constructed according to Algorithm 1 as follows, a sample or "height" under the distribution, h^i, is drawn uniformly from the interval $(\,0,\ f(\,\beta^{i-1}\,)\,)$. This height, h^i, defines a horizontal slice across the target density,

1. Sample $h \sim \text{Uniform}\{\, 0,\, f(\,\beta\,)\,\}$

2. Sample $\beta \sim$ Uniform on $A = \{\, \beta : f(\,\beta\,) \geq h \,\}$

Algorithm 1: Univariate Slice Sampling Algorithm for Parameter β

$A = \{\, \beta : f(\,\beta\,) \geq h^i \,\}$, which is then sampled from in step 2. In the univariate case, the slice A is the union of possibly several disjoint intervals. Neal (2003a) suggested two methods, stepping out and doubling, to construct the set A, for the single dimension case. In both methods, an interval is constructed to approximate the set A. To do so, an initial interval width is randomly oriented around the starting location β^{i-1}. The lower bound L^i and upper bound U^i are examined, and if either $f(\,L^i\,)$ or $f(\,U^i\,)$ is above the sampled height h^i, then the interval is extended. Once the interval is constructed, a new location β^i is selected from $(\,L^i,\, U^i\,)$ provided $\beta^i \in \{\, \beta : f(\,\beta\,) \geq h^i \,\}$. The sample h^i is then discarded, a new sample h^{i+1} is drawn, and the process repeats. The resulting Markov Chain has the desired stationary distribution (cf. Neal (2003a)).

In the step-out method, the lower bound is examined first and extended in steps equal to the initial interval width (ω) if it is above the sampled height h^i. The upper bound is then examined and extended. Once the interval is constructed, a shrinkage procedure is recommended after failed proposals to maximize sampling efficiency. When a proposed location $\widetilde{\beta}$ is drawn from $(\,L^i,\, U^i\,)$, but falls outside the target slice ($f(\widetilde{\beta}) < h^i$), the interval can be reduced in size. If $\widetilde{\beta} < \beta^{i-1}$, then set $L^i = \widetilde{\beta}$. Likewise, if $\widetilde{\beta} > \beta^{i-1}$, set $U^i = \widetilde{\beta}$. In this way, the interval collapses on failed proposals. Given that the current location must be within the slice, the probability of drawing a point from the slice then increases after each rejected proposal. (For details on the doubling method, see Neal (2003a))

While the univariate slice sampler is easy to implement and has many good theoretical properties, it can perform poorly in many cases, particularly when parameters are highly correlated. We illustrate this with the following simple two dimensional example.

Example 1 (Linear Regression with an Intercept) *Consider a simple lin-*

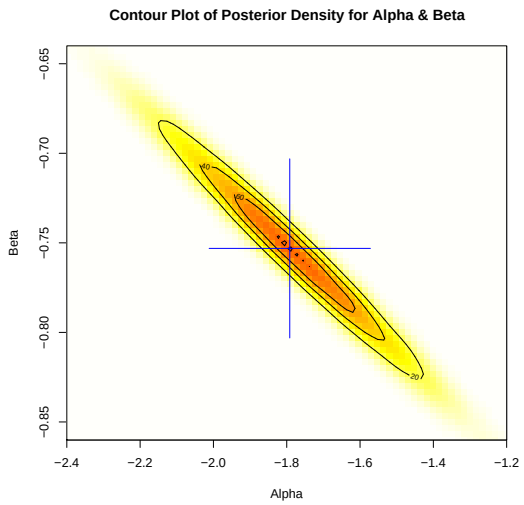

Figure 2.1. Joint Posterior Density of the intercept (α) and the regression coefficient (β) from Example 1

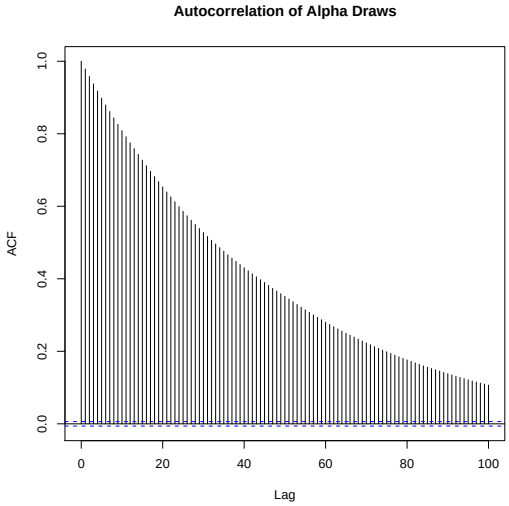

Figure 2.2. Autocorrelation for intercept parameter (α) using a Univariate Slice Sampler from Example 1

ear regression model. We assume that the errors ϵ_i are normally distributed with known variance of 1 *and are independent:*

$$Y_i = \alpha + \beta X_i + \epsilon_i \qquad \epsilon_i \sim N(0,1) \qquad i = 1, \ldots, N$$

We complete the Bayesian model specification by placing uniform priors on the intercept α and the regression coefficient β.

Gilks et al. (1996) noted that for this example, the posterior correlation of α and β is given by

$$\rho_{\alpha\beta} = -\frac{\overline{x}}{\sqrt{\overline{x}^2 + \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2}}$$

Therefore, by the above formula, we can control the mean and variance of the predictor in our simulations to arbitrarily fix the posterior correlation of the α and β parameters. Selecting a mean of 5.0 and a variance of 0.5, we generated a dataset with $\rho_{\alpha\beta} = -0.989391$.

MCMC sampling methods may be compared on the basis of effective sample size (ESS) and effective samples per second (ES/sec) as described by Kass et al.

(1998) and Chib and Carlin (1999). ESS is defined for each parameter as the total number of samples generated divided by the autocorrelation time τ, given by:

$$\tau = 1 + 2\sum_{i=1}^{N} \rho(k)$$

where $\rho(k)$ is the autocorrelation at lag k. The summation is usually truncated when the autocorrelation drops below 0.1, though more sophisticated approaches are possible (cf. Geyer (1992)). ESS is a rough estimate of the number of iid draws that are equivalent to the samples drawn.

Table 2.1. Comparison of algorithms for α and β from Example 1. All algorithms were run for $5,000,000$ samples.

	Intercept α		Coefficient β	
Algorithms	ESS	ES/sec	ESS	ES/sec
Step-out SS	53825	1231	56766	1298
Random-Walk MH	12853	932	13480	933
Multivariate SS	4312529	18400	5000000	21333

For comparison, we ran three algorithms to generate samples from the posterior distributions of α and β. The step-out slice sampler was run with interval widths of $\omega(\alpha, \beta) = (0.1, 0.02)$. A standard univariate random-walk Metropolis Hastings sampler was run with proposal variances: $\sigma(\alpha, \beta) = (0.0067, 0.00026)$. Finally, the multivariate slice sampler, outlined in Section 2.2, is included here for comparison. It was run with interval widths of $\omega(\alpha, \beta) = (0.074, 0.015)$. The interval widths and proposal variances were tuned to maximize ES/sec. The highly correlated posterior distribution of α and β which we plotted using $5,000,000$ samples of the univariate step-out slice sampler is shown in Figure 2.1 (true parameter values are marked by the $+$ symbol). It is evident in the slow decay on the autocorrelation plot (Figure 2.2) that the univariate slice sampler requires several iterations to move from one end of the density to the other. Examining the actual draws, we

noted that the sampler would often sit in one tail for a few hundred iterations or more before traversing the entire way across the distribution.

In comparing the performance of the three samplers, we see that the multivariate slice sampler is the clear winner in both ESS and ES/sec. This is apparent from Table 2.1 where we compare the effective sample size(ESS) and effective samples per second(ES/sec) for the univariate slice sampler, the univariate random-walk Metropolis-Hastings sampler, and the multivariate slice sampler (outlined in Section 2.2). Both univariate samplers exhibit a high autocorrelation and therefore a low ESS. The multivariate slice sampler, with much better mixing properties, exhibits a much lower autocorrelation and therefore a high ESS. Given the very inexpensive likelihood evaluations, the multivariate slice sampler also posts a high ES/sec.

In this toy example, the challenges posed by the collinearity of α and β can be easily remedied by standardizing the predictors (X_i). However, in complex hierarchical or non-linear models, removing this collinearity through transformations may be very difficult or impossible. In Section 2.3, we will examine a Gaussian process model where transformations to remove the posterior correlation between κ, ψ, and ϕ are not apparent. Neal (2003a) also provided a funnel-shaped example density where the univariate slice sampler performs poorly because the optimal interval width changes for different regions of the density. Several other creative univariate methods have been proposed such as the polar slice sampler of Roberts and Rosenthal (2002). The authors present an interesting example where the sampler's convergence is shown to be insensitive to the dimension of the target distribution. However, as Neal (2003b) mentions, the polar slice sampler requires fixing an origin for the polar coordinates and this may be difficult with little or no prior knowledge of the distribution.

The problems posed by strong dependence among parameters may be mitigated by using a multivariate slice sampler, which is much more adept at navigating complicated density functions. It is well established that multivariate sampling of blocked parameters can greatly outperform intelligent component-at-a-time updates (c.f. Liu et al. (1994); Roberts and Sahu (1997)). The next section will focus on the construction of a multivariate slice sampler.

2.2 Multivariate Slice Sampling

In contrast to the univariate slice sampler outlined in Section 2.1 which can sample a multivariate distribution in a component-at-a-time fashion, the multivariate slice sampler outlined below augments the multi-dimension parameter space with a single auxilliary variable. In the multivariate case, the approximate slice $\mathbf{A}$ forms a k-dimensional hypercube which bounds the target slice. Thus, the algorithm for the multivariate slice sampler is identical to Algorithm 1, except that the parameter β is now a vector and the interval A is now a k-dimensional hypercube $\mathbf{A}$.

As we did in the one-dimensional case, approximating the slice $A = \{\ \beta : f(\beta) \geq h\ \} \in \mathbb{R}^1$ by an interval (L, U), we can construct a k-dimensional hypercube to bound the slice $\mathbf{A} = \{\ \boldsymbol{\beta} : f(\boldsymbol{\beta}) \geq h\ \} \in \mathbb{R}^k$. As before, we begin by drawing a sample h^i uniformly from the interval $(0, f(\boldsymbol{\beta}^{i-1}))$ where $\boldsymbol{\beta}^{i-1}$ is now a k-dimensional vector. Then, an initial interval width is randomly oriented around the starting location $\boldsymbol{\beta}_j^{i-1}$ in each vector component. Then vertices of the hypercube, which we will refer to as the lower bound vector $\mathbf{L}^1$ and the upper bound vector $\mathbf{U}^1$, are examined. If the density evaluated at any vertices falls below the sampled height h^1, then the hypercube is expanded. Once the hypercube is constructed, a new location $\boldsymbol{\beta}^1$ is sampled provided $\boldsymbol{\beta}^1 \in \{\ \boldsymbol{\beta} : f(\boldsymbol{\beta}) \geq h^1\ \}$. The sample h^1 is then discarded, a new sample h^2 is drawn, and the process repeats.

The step-out algorithm does not generalize easily to the multivariate slice sampler because, from a computational standpoint, a k-dimensional hypercube has 2^k vertices, so the work doubles for each additional dimension considered. Also, in shrinking the hypercube in the obvious way (when proposals fall outside the slice), shrinking all dimensions performs poorly when the density does not vary rapidly in some dimensions (see Neal (2003a)). As the dimensionality of the target distribution increases, the k-dimensional hypercube is more likely to waste space, and consequently, the performance of rejection sampling for the proposal step will deteriorate. We chose to overcome this issue by parallelizing the evaluation of batches of proposals. Other approaches for constructing multivariate slice updates have also been proposed (see Neal (2003a)), but we find that our approach is relatively simple and easily lends itself to parallel processing.

2.3 Application to Gaussian Process Models

In Section 2.1, we compared the performance of univariate and multivariate sampling algorithms in the context of a toy example. Example 1 was chosen to highlight the shortcomings of a univariate sampling algorithm (slice sampler or otherwise) when applied to a model parameterization with a highly correlated posterior distribution. However, the likelihood evaluations are computationally inexpensive, making it far less challenging than many commonly used Bayesian models. To examine the performance of the multivariate slice sampler within the context of a more realistic example, we turn to the linear Gaussian process model.

Example 2 (Linear Gaussian Process Model) *Consider a linear Gaussian process model with an exponential covariance function, a very popular model for spatial data (cf. Cressie (1993), Banerjee et al. (2004)). We model a spatially-referenced response $Y(\mathbf{s}_i)$ measured at a locations s_i by a set of covariates $X(\mathbf{s}_i)$ ($i \in \{1 \dots N\}$). The responses at the locations are correlated based on magnitude of separation and this falls off at an exponential rate.*

$$Y(\mathbf{s}_i) = X(\mathbf{s}_i)\boldsymbol{\beta} + \epsilon(\mathbf{s}_i) \qquad \epsilon(\mathbf{s}_i) \sim N(0, \Sigma(\mathbf{s}))$$

where the covariance matrix $\Sigma(\mathbf{s}_i, \mathbf{s}_j)$ is parameterized as:

$$\Sigma(\mathbf{s}_i, \mathbf{s}_j) = \begin{cases} \kappa + \psi & i = j \\ \kappa \exp\left(\frac{||\mathbf{s}_i - \mathbf{s}_j||^2}{\phi}\right) & i \neq j \end{cases}$$

We place a uniform prior on $\boldsymbol{\beta}$ as in Example 1. We place inverse gamma (shape = 2, scale = 1) priors on κ and ψ so that the prior means for κ and ψ are 1.0 and the prior variance is infinite. We place a uniform prior on the effective range parameter ϕ with a lower bound of 0.01 and an upper bound of 5.0.

The computational complexity of this model is easily controlled by the number of locations N included in the dataset. To compare the univariate and multivariate samplers across a range of complexities, we generated five datasets with 100, 200, 300, 400, and 500 locations. Each dataset was generated from model outlined in Example 2 with $\psi = \kappa = 1.0$ and $\phi = 0.2$ as the true values. In Table 2.2, all of

the reported effective sample sizes (and those per second) reflect a simulation of $10,000$ iterations. We found this to be sufficient in most cases to meet the minimum benchmark of 1000 effective samples. As the comparison of the algorithms is based on sampling efficiency, we chose to start all samples at the true values so as to avoid questions of convergence and appropriate length of burnin period. We will address these issues within the context of a real data set in Section 4.3.

When implementing the univariate and multivariate slice samplers, we chose to use the step-out method for constructing the approximte slice $\mathbf{A}$. Mira and Roberts (2003) note that the step-out method is unable to move between two disjoint modes that are separated by a region of zero probability (larger than the step size). This implies that the sampler may not be irreducible for multimodal distributions where the initial step size is too small; however, this problem does not arise in any of the examples considered here.

In Table 2.2, we see that the multivariate slice sampler provides a significant improvement in ESS over the univariate slice sampler for the parameters κ and ψ. As the standard multivariate slice sampler we have implemented uses a rejection sampler for the proposals, it is not surprizing that the effective sample size of ϕ went down. The penalty of adding more dimensions is that those parameters which mix well will be less able to do so with the addition of poorly mixing parameters.

2.4 Motivation for Exploring Parallelism

The multivariate slice sampler described above can make large moves across the target density and explore it with reasonably efficiency. Under reasonably weak conditions, Roberts and Rosenthal (1999) showed that slice sampler is nearly always geometrically ergodic - a convergence criterion that a generically applicable random walk sampler often fails to meet. Further, Mira and Tierney (2002) provide a sufficient condition under which they show that the slice sampler is uniformly ergodic.

However, as in the univariate case, the real challenge is to appropriately construct the slice $\mathbf{A} = \{\boldsymbol{\beta} : f(\boldsymbol{\beta}) \geq h\}$. Often, the computational challenge of evaluating all boundary points of a k-dimensional hypercube may drag the performance of the multivariate slice sampler well below the performance of the univariate meth-

Table 2.2. Comparison of single-threaded slice samplings algorithms for κ, ψ, and ϕ from Example 2. All algorithms were run for $10,000$ iterations.

Slice Sampler Algorithm		Number of Locations: 100 ESS	100 ES/sec	200 ESS	200 ES/sec	300 ESS	300 ES/sec	400 ESS	400 ES/sec	500 ESS	500 ES/sec
Univariate	κ	1490	(21.21)	657	(2.07)	522	(0.59)	483	(0.29)	336	(0.10)
	ψ	1552	(22.08)	644	(2.03)	516	(0.58)	490	(0.29)	338	(0.10)
	ϕ	3127	(44.49)	3709	(11.70)	3585	(4.02)	3295	(1.96)	5559	(1.69)
Multivariate	κ	5207	(40.30)	3251	(4.80)	3051	(1.33)	3395	(0.85)	2519	(0.28)
	ψ	4782	(37.02)	3374	(4.98)	3028	(1.32)	3482	(0.87)	2576	(0.29)
	ϕ	2160	(16.72)	3157	(4.66)	3535	(1.54)	1467	(0.37)	4657	(0.52)

Note: Here we compare the effective sample size(ESS) and effective samples per second(ES/sec) for the univariate and multivariate slice samplers. We see that even in analyzing the more computationally expensive linear Gaussian process model of Example 2, the multivariate slice sampler's ESS and ES/sec are higher than the univariate slice sampler.

ods. For challenging density functions, each additional evaluation of the likelihood can take several seconds, several minutes, or longer. An algorithm which requires even a few expensive likelihood evaluations quickly becomes computationally infeasible. In Section 2.3, we provided simulation results for a Gaussian process model where additional likelihood evaluations significantly slow down computation (see Example 2). On the other hand, since the likelihood evaluations can be done independently of each other, they can easily be parallelized. In heavily parallelized computing, all of the needed likelihood evaluations can take place in little more than the time needed for a single evaluation in a serial environment.

Further, the benefit of using a multivariate slice sampler is that the additional likelihood evaluations are used in an optimal way to inform the algorithm of the multivariate shape of the target distribution, thereby constructing a sampler which is better able to traverse quickly to all regions of the density. As we will see in Section 4, a multivariate slice sampler can much more effectively utilize parallel likelihood evaluations. Clearly, as the dimensionality and hierarchical complexity of the desired model increases, the generality of the parallel multivariate slice

sampler will affords us an intelligent sampler, capable of providing an accurate picture of the target density in a reasonable amount of time.

Chapter 3

Parallel Computation

In searching for ways to parallelize an MCMC sampler, we attempt to identify portions of the algorithm which are independent and hence can be computed simultaneously. The multivariate slice sampler outlined in the previous section requires several likelihood evaluations per iteration. The selection of an appropriate statistical model and parameterization for a given problem will have a major impact on the ability to parallelize computations. In a Bayesian setting, examining the full conditional distributions may lead to opportunities to parallelize the model. Often natural groupings will arise where, for example in mixture distributions, the coefficients for the first component can be updated independently of a second component. Several authors have discussed the decomposition of a given parameterization into maximal number of sets which can be updated independently. (cf. Whiley and Wilson (2004), Byrd et al. (2008), and Yan et al. (2007)) However, in many statistical models, the structure of the resulting distributions may preclude strategies that take advantage of conditional independence as is the case with the linear Gaussian process model (see Example 2). In the next sections, we will briefly describe two software packages which can be used to parallelize MCMC algorithms. These two packages, OpenMP and CUDA, are only a small sample of several packages available for parallel computing. We now provide a brief overview of OpenMP and CUDA and how they can be utilized to parallelize MCMC samplers.

3.1 OpenMP

Open Multi-Processing (OpenMP) is a specification with the basic purpose of providing a generic interface to support shared memory multiprocessing. For the purpose of statistical computation, the power of OpenMP lies in its ease of implementation. The addition of a single compiler directive will transform a non-threaded "for loop" and separate it across several CPU cores within the same system. The major limitation of OpenMP is not truly in the software specification, but in the hardware for which it is designed. Current multi-core, multi-processor shared memory systems can scale with the dimension of a given problem but only at a huge expense. Most multi-core, multi-processor systems have only a dozen or so CPUs which greatly limits the practical speedup one can acheive. The most promising platform for OpenMP is Intel's upcoming Larabee chipset, due out in late 2009, which promises 32 and 48 cores in the first and second versions respectively. Yet taking into account these limitations, OpenMP lends utility to problems of moderate size due to the incredible ease of implementation (a single line of code) and a reasonable improvement in speed attained through parallel computation.

OpenMP is designed for multithreaded processing within a single computer (with one or more multicore chips). It does not support communication between computers. A different specification known as the Message Passing Interface (MPI), is designed to overcome this limitation. MPI allows for processing with a cluster of computers. Parallel implementations of simple matrix algorithms (multiplication, Cholesky decomposition, etc.) are available in PLAPACK (Using MPI) and ScaLAPACK(Using OpenMP or MPI). The parallel spatial models of Yan et al. (2007) demonstrated that PLAPACK based block matrix algorithms provide similar factors of improvement to the OpenMP results described below. As the focus of this paper is primarily on software which runs on a single physical machince (with a few processors / cores), we have not run simulations which use MPI. However, in the future, we envision combining these technologies where MPI is used to distribute likelihood evaluations across several computers, each of which is equipped with multiple graphics cards. Then OpenMP and CUDA would be used cooperatively to evaluate a single likelihood function within each machine in the cluster using multiple CPUs and multiple GPUs.

3.2 CUDA

Compute Unified Device Architecture (CUDA), introduced in 2006, is a C/C++ interface to the vectorized processing available on a graphics processing unit (GPU). A GPU differs significantly from a traditional CPU in a few key ways. First, creating and destroying threads on a CPU is very expensive, often requiring thousands of cycles or more, but on a GPU, one can create several thousand threads in less than ten clock cycles. The second key difference is that on a CPU most instructions work with a single set of operands: adding two numbers, multiplying two numbers, etc. On a GPU, all instructions use vectors of operands - they add two vectors, multiply two vectors, etc. These two differences allow a program which could only use a few CPU threads efficiently to instead utilize $30,000$ or more GPU threads and attain a level of parallelism which was previously unthinkable.

To briefly overview terminology, a set of instructions which CUDA executes on a GPU is known as a kernel. When a kernel is launched from a parent CPU program, in addition to functional inputs (e.g. parameter values), the kernel must be specified with the number of independent blocks of parameters to process and the number of threads which will work cooperatively on each block. In the context of the multivariate slice sampler of Section 2.2, the kernel function calculates the likelihood at a given location. The number of blocks will differ between the slice construction step and the proposal step. (For details, see Section 4.1). The number of threads per block will vary based on the size and complexity of the likelihood calculation. The GPU used in Section 4, a GTX 280, is organized into 30 multiprocessors, each of which is capable of executing 1024 threads simultaneously (512 per block) in one, two, or three dimension grids. In Examples 2 and 3, we will never evaluate more than 60 blocks at once; hence, to maximize both throughput and the usage of the GPU, we will allocate 512 threads per block. In Section 4.1 we describe a small set of trial runs that were used to determine the optimal dimensions of the thread block.

In Section 4 we return to the Gaussian process example from Section 2.3. OpenMP proves to be quite simple to implement and tune. The CUDA based sampler requires more tuning to acheive optimal results but this is mostly because with greater than $30,000$ threads available we chose to parallelize the matrix com-

putations in addition to batching the likelihood evaluations. Both OpenMP and CUDA show utility in parallelizing the multivariate slice sampler - OpenMP for its ease of use and CUDA for its immense processing power.

Chapter 4

A Parallel Implementation of Multivariate Slice Samplings

In this section, we return to the spatial Gaussian process model presented in Section 2.3. As mentioned above, the multivariate slice sampler invovles multiple likelihood evaluations at each iteration of the algorithm. Here we provide results of running parallel multivariate slice sampler using both OpenMP and CUDA on the same datasets generated in Section 2.3. In Section 4.1, we briefly outline the tuning and implementation details for the OpenMP and CUDA based samplers. Then in Section 4.2, we compare the results of the parallel samplers with the univariate slice sampler. Finally, in Section 4.3, we turn to the analysis of a real dataset. As we shall see in the sections that follow, the results simultaneously demonstrate both the utility of the multivariate slice sampler and the improvement in speed gained through parallelization.

4.1 Implementation Details

To parallelize the multivariate slice sampler for Examples 2 and 3, we focused on three pieces of the algorithm: the slice construction, the proposals, and the likelihood function itself. First, when constructing the approximate slice $\mathbf{A}$ using a three dimensional hypercube (shown in Algorithm 2), the likelihood at each of the 8 hypercube vertices can be evaluated independently. After evaluating the likelihoods, should we need to step out in any dimension, we then re-evaluate the

updated hypercube vertices in parallel. Second, when proposing a new location (shown in Algorithm 3), we chose to use a simple rejection sampler and evaluate batches of proposals in parallel. The first proposal to fall within the target slice is then accepted and the parameters are updated. For the OpenMP sampler, we set the proposal batch size equal to the number of threads (either three or four). For the CUDA sampler, we set the proposal batch size equal to 30 (the number of blocks that allowed for maximal throughput). Finally, in the CUDA-based multivariate slice sampler, we also chose to parallelize the matrix operations in the likelihood function itself.

$\boldsymbol{\beta}$ the current parameter value ($\boldsymbol{\beta} \in \mathbb{R}^k$)	Sample $h \sim \text{Uniform}\{\, 0,\, f(\,\boldsymbol{\beta}\,)\,\}$
ω the initial interval widths for constructing a hyperrectangle in k dimensions.	For $j \in \{1 \ldots k\}$ $u \sim \text{Uniform}(0, 1)$ $\mathbf{L}_j = \boldsymbol{\beta}_j - \omega_j * u$ $\mathbf{U}_j = \mathbf{L}_j + \omega_j$
$\mathbf{L}$ Lower bounds of hypercube	Do Construct list of k dimensional hypercube vertices from $\mathbf{L}$ and $\mathbf{U}$
$\mathbf{U}$ Upper bounds of hypercube	Evaluate 2^k vertices in parallel
f function proportional to the full conditional distribution for $\boldsymbol{\beta}$.	Test each hypercube vertex, if $(h < f(vertex))$ extend $\mathbf{L}$ and/or $\mathbf{U}$ by ω as appropriate
	While ($\mathbf{L}$ or $\mathbf{U}$ changed)

Algorithm 2: A parallel step-out procedure for constructing an approximate multivariate slice for the parameter $\boldsymbol{\beta}$.

OpenMP

Before we can compare the relative efficiency of the parallel multivariate slice sampler using OpenMP with the other samplers, we must first run a short tuning study to determine the optimal number of threads. We expect that the optimal number of threads will depend on the number of locations in the model, as the

$\boldsymbol{\beta}$	the current parameter value
$\mathbf{L}$	Lower bounds of hypercube
$\mathbf{U}$	Upper bounds of hypercube
f	function proportional to the full conditional distribution for $\boldsymbol{\beta}$.

Repeat until proposal not rejected

  Draw batch of N proposals uniformly from the hypercube bounded by $\mathbf{L}$ and $\mathbf{U}$

  Evaluate N proposals in parallel

  On first proposal m such that $(h < f(proposal[m]))$

    Update $\boldsymbol{\beta} = proposal[m]$ and stop

Algorithm 3: A batched parallel rejection sampler for the multivariate slice sampler update of the parameter $\boldsymbol{\beta}$.

computational expense of the Gaussian process model likelihood function depends directly on the Cholesky decomposition (Order $N^3/3$). To allow the results to be directly comparable to Section 2.3, we ran the same simulations on the same datasets for the same number of iterations $(10,000)$. But in our experience, trial runs of 400 samples or less are sufficient to determine the best threading model. We found that for 100 to 300 locations, three OpenMP threads was optimal. For 400 to 500 locations, four OpenMP threads was optimal. Clearly the geometry and configuration of the processor hardware will have an impact on the speed of execution and the realized speedup (or lag) of using additional processor cores. The simulations were run on a dual quad-core E5430 Xeon system. There is a natural penalty in performance when moving from four threads which can run on a single chip to five threads which must run on two chips. It is not surprising that in our tuning study, we found that adding a fifth thread dropped the computation efficiency by as much as 10%.

There are also a number of second order effects which will impact the speedup of the multivariate slice sampler using OpenMP. For example, the default memory allocation mechanism may be thread safe(depending on the choice of compiler and hence standard C++ library), but it is often very inefficient in a multithreaded environment. The results presented in Table 4.1 were obtained after incorporating the NEDMalloc software of Douglas (2008). NEDMalloc is a multithreaded allocator designed to reduce the mutex based locking of a single threaded (threadsafe) allocator. As expected, the improvement in speed (30% at best) was larger as the number of threads increased.

In the results provided in Section 4.2, we see that paralellizing the multivariate slice sampler with OpenMP attains only a marginal speedup. This is due to the moderate expense of the Gaussian process model likelihood function for these relatively small choices of model size. We chose 500 locations as an upper bound due the hardware register constraints within the graphics card. Clearly as the location count increases the workload will increase. Hence, the OpenMP threads will be better utilized and that will yield a higher relative speedup. Finally, though the relative speedup using OpenMP may not be large, the burden of adding one or two additional lines of code should be weighed against the 40% performance improvement.

CUDA

As with the OpenMP based sampler, we chose to parallelize the evaluation of the hypercube vertices in Algorithm 2 and also the batched proposal evaluations in Algorithm 3. But with the advantage of about $30,000$ threads to work with in the CUDA based sampler, we also chose to parallelize the evaluation of each likelihood computation. The evaluation of the full conditional distribution for κ, ψ, and ϕ, given below in Equation (4.1), can be decomposed into five major steps, outlined in Algorithm 4. Note that the bulk of the computational expense occurs in the the lower triangular Cholesky factorization in step two.

$$\begin{aligned}\pi(\kappa,\psi,\phi|-) \propto |\Sigma|^{-1/2} \exp\left\{-\frac{1}{2}\left(Y-X\boldsymbol{\beta}\right)\Sigma^{-1}\left(Y-X\boldsymbol{\beta}\right)\right\} \quad \times \\ \kappa^{-(\alpha_\kappa+1)} \exp\frac{-\beta_\kappa}{\kappa} \psi^{-(\alpha_\psi+1)} \exp\frac{-\beta_\psi}{\psi}\end{aligned} \tag{4.1}$$

The evaluation of the full conditional distribution for κ, ψ, and ϕ within the CUDA kernel can be parallelized in each step of Algorithm 4. In step one, the construction of Σ is done column by column. By virtue of having more threads (512) than locations (at most 500), each thread is assigned to compute a single element in the current column of $\Sigma_{i,j}$. Then that column of $\Sigma_{i,j}$ is immediately used to compute the lower triangular Cholesky factorization via a standard block decomposition (cf. Golub and Van Loan (1996)). Hence, storage of more than one

1. Construct the covariance matrix Σ from the given κ, ψ, and ϕ values and the list of locations s_i. Recall that Σ is defined as follows:

$$\Sigma(\mathbf{s}_i, \mathbf{s}_j) = \begin{cases} \kappa + \psi & i = j \\ \kappa \exp\left(\frac{||\mathbf{s}_i - \mathbf{s}_j||^2}{\phi}\right) & i \neq j \end{cases}$$

2. Compute the lower triangular Cholesky factorization of Σ.
3. Solve for $\Sigma^{-1}(Y - X\boldsymbol{\beta})$ via forward and back substitution.
4. Compute the dot product of $(Y - X\boldsymbol{\beta})$ with the results of step #3.
5. Compute the determinant of Σ and the other remaining terms.

Algorithm 4: An algorithmic representation for the evaluation of the full conditional distribution of κ, ψ, and ϕ in Examples 2 and 3.

column of Σ is not required. In step three of Algorithm 4, both the forward and back substitution are performed using one column and one row, respectively, at a time and again, each element in the vector is given to a separate thread. During the back substitution, each thread also simultaneously computes that elements contribution to the dot product for step four and the diagonal element's contribution to the determinant in step five. Hence, upon completion of the back substitution, steps four and five are virtually finished, leaving only the contribution of the priors to be computed.

As CUDA thread-blocks can be allocated in two or three dimension grids, thereby reducing the number cycles spent calculating indices, the CUDA based sampler did require a small number of trial runs to determine the optimal layout of the two dimensional thread-block. We used a batch of 200 likelihood evaluations to test the various block dimensions. We found that a 64×8 block size was optimal for all model sizes. Note that two dimensional indexing was only used for the block Cholesky factorization. The forard and back substitution and other matrix operations used linear thread indexing. There are other hardware constraints (shared memory capacity, memory transfer penalties, etc.) which are beyond the scope of the discussion here, but were taken into account in designing the CUDA-based likelihood evaluation. However, it should be noted that nVidia

has provides a CUDA Occupancy calculator which, when given the shared memory requirements and register usage of the algorithm, will provide the optimal thread count and thereby mostly circumvent the need to run an exhaustive grid search to tune the sampler.

As mentioned in the previous section, the current GPU hardware and software (CUDA 2.2) limits the maximum thread block size to 512 threads. As such, we did not test models with greater than 500 locations. As new hardware is developed which supports larger block sizes, we hope to attain better improvements for larger models. Future research will also focus on utilitizing multiple GPUs at a time. Current technology limits one to using a maximum of four GPUs simultaneously, but this would allow for 960 simultaneously likelihood evaluations.

4.2 Summary of Simulation Study

Table 4.1 contains the results of a simulation based on Example 2 using the univariate slice sampler, the single-threaded multivariate slice sampler, the optimal OpenMP multivariate slice sampler, and the optimal CUDA multivariate slice sampler. In examining Table 4.1, we note first that for all of the samplers shown, the ES/secdecreases as the number of locations increase. The ES/secof κ and ψ for the 100 location model is roughly ten times that of the 200 location model for all of the samplers. The dependence of κ and ψ is also fairly evident in Table 4.1. In simulating these datasets, we selected a true range parameter ϕ of 0.2. As such, the spatial dependence decays fairly quickly and consequently, this induces a strong dependence between κ and ψ because the diagonal elements of Σ are equal to $\kappa + \psi$.

As was discussed in Section 2.3, even the relatively slow single CPU multivariate slice sampler does outperform the univariate slice sampler in ES/secfor κ and ψ mainly because of a much larger ESS. However, the ES/secfor ϕ for the single CPU multivariate slice sampler is much lower than that of the univariate slice sampler. Here again, this is due to the moderate spatial depenence. By blocking the update of ϕ with κ and ψ in a three dimensional sampler, the ability to mix across the distribution of ϕ is greatly hampered by the strong dependence between κ and ψ. However, the ES/secof κ and ψ is still clearly the limiting factor.

Practically speaking, it is the smallest ES/secwhich determines the minimal run time of a given algorithm. In other words, to produce at least $10,000$ effective samples for κ, ψ, and ϕ in the 300 location model using the univariate slice sampler one would need to run the sampler for $10000/0.58 = 17,240$ seconds (roughly 5 hours). To generate the same $10,000$ effective samples for κ, ψ, and ϕ using the single CPU multivariate slice sampler, one would need to run the sampler for $1000/1.32 = 7,575$ seconds (2.1 hours). However, when we examine the 400 location model, this factor of two improvement nearly vanishes because the ratio of the smallest multivariate ES/sec(ϕ ES/sec$= 0.37$) to the smallest univariate ES/sec(κ ES/sec$= 0.29$) is only 1.27. Hence, the net gain of using a more complicated algorithm is only an improvement of 27%. However, the picture changes quite dramatically when we examine the parallelized samplers.

The OpenMP based sampler shows a reasonable 40 to 50% improvement in ES/secon top of the improvement that the multivariate slice sampler makes in ES/secover the univariate slice sampler. The largest improvement occurs in the 300 location model, but even in this case, the ϕ ES/secis still only 57% of the univariate slice sampler's ES/sec. However, the ratio of the smallest OpenMP multivariate ES/sec(ϕ ES/sec$= 0.52$) to the smallest univariate ES/sec(κ ES/sec$= 0.29$) is 1.79. Hence, the speedup of the algorithm directly translates to shorter run times.

The CUDA-based multivariate slice sampler makes an even more compelling argument. In Table 4.1, we note first that the speedup factor increases significantly as the size of the model increases. In the 500 location model, the CUDA based multivariate slice sampler is 5.4 faster than the single multivariate slice sampler CPU version. When we examine the sampling times, the univariate slice sampler would require $10000/.10 = 100,000$ seconds (roughly 28 hours) to produce $10,000$ effective samples for all paramters; whereas, the CUDA based multivariate slice sampler would require only $10000/1.51 = 6622$ seconds (or roughly 1.8 hours). Hence, the CUDA based multivariate slice sampler improves upon the efficiency of the univariate slice sampler by a factor of roughly 15.

After examining the results of Table 4.1, we did investigated using a two dimensional multivariate slice sampler for only κ and ψ, but found that the added cost of a univariate update for ϕ pulled the performance of the multivariate sampler down dramatically, especially in the CUDA based implementation. This highlights

Table 4.1. Comparison of effective samples per second and relative speedup for single and multi-threaded slice samplings algorithms for κ, ψ, and ϕ from Example 2. All algorithms were run for $10,000$ iterations.

Slice Sampler Algorithm		Number of Locations									
		100		200		300		400		500	
Univariate Single CPU	κ	21.21		2.07		0.59		0.29		0.10	
	ψ	22.08		2.03		0.58		0.29		0.10	
	ϕ	44.49		11.70		4.02		1.96		1.69	
Multivariate Single CPU	κ	40.30	(1.90)	4.80	(2.31)	1.33	(2.27)	0.85	(2.96)	0.28	(2.73)
	ψ	37.02	(1.68)	4.98	(2.45)	1.32	(2.28)	0.87	(2.99)	0.29	(2.78)
	ϕ	16.72	(0.38)	4.66	(0.40)	1.54	(0.38)	0.37	(0.19)	0.52	(0.31)
Multivariate OpenMP	κ	56.73	(2.67)	6.83	(3.30)	1.96	(3.34)	1.19	(4.14)	0.37	(3.59)
	ψ	52.10	(2.36)	7.09	(3.49)	1.94	(3.35)	1.22	(4.20)	0.38	(3.65)
	ϕ	23.53	(0.53)	6.64	(0.57)	2.27	(0.56)	0.52	(0.26)	0.68	(0.40)
Multivariate CUDA	κ	116.77	(5.51)	19.21	(9.26)	6.16	(10.51)	3.72	(12.94)	1.51	(14.78)
	ψ	119.83	(5.43)	19.19	(9.44)	6.31	(10.88)	3.90	(13.38)	1.53	(14.84)
	ϕ	64.56	(1.45)	17.57	(1.50)	8.61	(2.14)	1.59	(0.81)	2.88	(1.70)

ES/sec (speedup)

Note: Table 4.1 uses the univariate slice sampler's run time (single-threaded CPU-based) as the baseline for determining algorithmic speedups shown above for Example 2 (c.f. Sections 2.3 and 4)

the fact that there is a negligible difference between estimating one, two, three, four, or five dimension multivariate slice samplers on the GPU because all will fit within the 60 block maximum (to acheive peak performance). Further, in situations which require higher dimensional slice samplers (up to 9D), four graphics cards could be combined using the CUDA API to compute all needed likelihood evaluations simultaneously.

Note that we have not included a study of parallelized univariate sampling. While it would be beneficial for both multivariate and univariate slice sampling to speed up or parallelize each individual likelihood calculation, the univariate slice sampler would not benefit nearly as much from the two parallelizations in

Algorithms 2 and 3. As the univariate slice sampler usually requires only three or four likelihood evaluations per iteration, a speedup factor of two or three gained by evaluating these likelihoods in parallel would only place the ES/sec on par with the single CPU multivarite slice sampler.

In the next section, we benchmark the univariate slice sampler, and the OpenMP and CUDA based multivariate slice samplers on a surface temperature dataset. We investigate the sample efficiency of the algorithms as well as the rate of convergence from random starting locations to the posterior mode.

4.3 Application to Surface Temperature Data

In the this section, we apply the linear Gaussian process model from Example 2 to the analysis of the mean surface temperature over the month of January, 1995 on a 24×21 grid covering Central America. These data were obtained from the NASA Langley Research Center Atmospheric Science Data Center.

Example 3 (ASDC Surface Temperature Dataset) *We wish to model the mean surface temperature $Y(\mathbf{s}_i)$ measured at 500 locations s_i by a set of covariates $X(\mathbf{s}_i)$ which includes an intercept and two covariates: the latitude and longitude of each grid point. We standardized the latitude and longitude to remove collinearity with the intercept. We assume that the surface temperatures are correlated based on the distance between locations and that this falls off at an exponential rate. Hence, we fit a linear Gaussian process model with an exponential covariance function:*

$$Y(\mathbf{s}_i) = X(\mathbf{s}_i)\boldsymbol{\beta} + \epsilon(\mathbf{s}_i) \qquad \epsilon(\mathbf{s}_i) \sim N(0, \Sigma(\mathbf{s}))$$

where the covariance matrix $\Sigma(\mathbf{s}_i, \mathbf{s}_j)$ is parameterized as in Example 2. We place a uniform prior on $\boldsymbol{\beta}$ as was done in Examples 1 and 2. We complete the Bayesian model specification, by placing inverse gamma (shape = 2, scale = 1) priors on κ and ψ and a uniform prior on ϕ with a lower bound of 0.1 and an upper bound of 50.0 (which is slightly larger due to the increased separation in the grid points).

For the analysis of the surface temperature data, we had to address the question of starting values and appropriate length of burnin. We tried several different

random starting values and found that all of the samplers converged incredibly quickly for this dataset. We initialized the coefficients β to zero plus random normal offset. We initialized ψ, κ, and ϕ, to a $gamma(3, 1)$ random variable plus 1.0 (to ensure that the parameters were strictly positive). The initial interval widths for all of the samplers were tuned in order to maximize ES/secby using a large grid search. The multivariate slice samplers converged to the support of the posterior densities within three samples. The univariate slice sampler took slightly longer, but even with very extreme values it never took more than 30 samples to reach the posterior mean. However, to virtually eliminate issues due to the choice of starting values, we report results in Table 4.2 from $100,000$ samples having first discarded an initial burnin run of $100,000$ samples.

In examining the posterior parameter estimates for this dataset, we have $\mu_\kappa = 55$, $\mu_\phi = 49$, and $\mu_\psi = 0.2$. In contrast to the simulation study of Section 4, this dataset exhibits a very strong spatial dependence. In Table 4.2 we see that the parameters κ and ϕ are strongly correlated; whereas κ and ψ were strongly correlated in the simulation study. But we find, in comparing the univariate slice sampler to the parallelized multivariate slice sampler that many of the same relationships from simulation study hold here as well. For example, we see that the ESSfor κ and ϕ of the multivariate samplers is more than ten times the ESSfor the univariate sampler. And though the multivariate slice sampler is much more computationally expensive, the parallelization through OpenMP and especially through CUDA is sufficient to mitigate the increased computational burden. By comparing the minimum ES/sec, we see that the minimum ES/secof the CUDA based multivariate slice sampler (κ ES/sec= 2.42) is roughly 13 times the minimum ES/secof the univariate slice sampler (ϕ ES/sec= 0.18). As in the simulation study described in Section 4, the CUDA based multivariate slice sampler provides a signficant improvement over the univariate slice sampler.

Table 4.2. Comparison of effective sample size (ESS), effective samples per second(ES/sec), and relative speedup of ES/sec for κ, ψ, and ϕ from Example 3. All algorithms were run for $200,000$ iterations, but the first $100,000$ were discarded to allow for sampler burnin.

	ψ			κ			ϕ		
Algorithms	ESS	ES/sec	ES/sec Speedup	ESS	ES/sec	ES/sec Speedup	ESS	ES/sec	ES/sec Speedup
Univariate Slice Sampler	65254	2.89		4284	0.19		4147	0.18	
Multivariate Slice Sampler (using OpenMP)	54640	1.86	(0.64)	52324	1.79	(9.42)	55306	1.89	(10.50)
Multivariate Slice Sampler (using CUDA)	54794	2.59	(0.90)	51177	2.42	(12.74)	54229	2.57	(14.28)

Note: Table 4.2 uses the univariate slice sampler's run time as the baseline for determining algorithmic speedups shown above for Example 3 (c.f. Section 4.3)

Chapter 5

Discussion

We have examined the performance of the univariate and multivariate slice samplers within the context of two examples and the surface temperature analysis. In our simulations, we found that the multivariate slice sampler was much more efficient than the univariate methods when the posterior distribution is highly correlated along one or more dimensions. However, the multivariate slice sampler is very computationally expensive per iteration, expecially when likelihood evaluations are expensive. But even in the context of a more expensive linear Gaussian process model, we found that the multivariate slice sampler's excellent mixing properties allowed it to best the univariate slice sampler's efficiency.

We then noted that the likelihood evaluations utilized in constructing an approximate slice could be independently evaluated; hence, hypercube vertices could be evaluated in parallel and we investigated two different threading implementations. The OpenMP-based implementation requires a very minimal modification to the code, but only attained a 40 to 50% improvement over the single-threaded multivariate slice sampler. In contrast, the CUDA-based implementation was five times faster than the single-threaded CPU-based implementation. When we combined this speedup with the already superior ESS of the multivariate algorithm, we found that the CUDA-based solution yielded a sampler which was 14 times more efficient than the single-threaded univariate slice sampler in both the simulation study of Section 4 and the surface temperature analysis of Section 4.3. Clearly, after being parallelized, the multivariate slice sampler is an efficient and viable alternative to the univariate slice sampler.

Bibliography

Agarwal, D. K. and Gelfand, A. E. (2005). Slice sampling for simulation based fitting of spatial data models. *Statistics and Computing*, 15(1):61–69.

Andrieu, C., de Freitas, N., Doucet, A., and Jordan, M. I. (2003). An introduction to mcmc for machine learning. *Machine Learning*, 50(1):5–43.

Banerjee, S., Carlin, B., and Gelfand, A. (2004). *Hierarchical modeling and analysis for spatial data*. Chapman & Hall Ltd.

Byrd, J. M. R., Jarvis, S. A., and Bhalerao, A. H. (2008). Reducing the run-time of mcmc programs by multithreading on smp architectures. In *IPDPS*, pages 1–8. IEEE.

Chib, S. and Carlin, B. P. (1999). On mcmc sampling in hierarchical longitudinal models. *Statistics and Computing*, 9:17–26.

Cressie, N. A. C. (1993). *Statistics for Spatial Data*. Wiley Series in Probability and Statistics. Wiley-Interscience, New York, 2nd. edition.

Damien, P., Wakefield, J., and Walker, S. (1999). Gibbs sampling for bayesian non-conjugate and hierarchical models by using auxiliary variables. *Journal of the Royal Statistical Society. Series B (Statistical Methodology)*, 61(2):331–344.

Douglas, N. (2008). Nedmalloc. http://www.nedprod.com/programs/portable/nedmalloc/.

Geyer, C. J. (1992). Practical markov chain monte carlo. *Statistical Science*, 7(4):473–483.

Gilks, W., Richardson, S., and Spiegelhalter, D. (1996). Estimation and optimization of functions. In Gilks, W. R., Richardson, S., and Spiegelhalter, D. J., editors, *Markov chain Monte Carlo in practice*, pages 241–258. London: Chapman & Hall/CRC.

Golub, G. H. and Van Loan, C. F. (1996). *Matrix Computations (Johns Hopkins Studies in Mathematical Sciences)*. The Johns Hopkins University Press.

Kass, R. E., Carlin, B. P., Gelman, A., and Neal, R. M. (1998). Markov chain monte carlo in practice: A roundtable discussion. *The American Statistician*, 52(2):93–100.

Kinney, S. K. and Dunson, D. B. (2007). Fixed and random effects selection in linear and logistic models. *Biometrics*, 63:690–698(9).

Kovac, K. (2005). Machine learning for bayesian neural networks. Master of science, University of Toronto.

Lewis, P. O., Holder, M. T., and Holsinger, K. E. (2005). Polytomies and bayesian phylogenetic inference. *Systematic Biology*, 54(2):241–253.

Liu, J. S., Wong, W. H., and Kong, A. (1994). Covariance structure of the gibbs sampler with applications to the comparisons of estimators and augmentation schemes. *Biometrika*, 81(1):27–40.

Mackay, D. J. C. (2002). *Information Theory, Inference & Learning Algorithms*. Cambridge University Press.

Mira, A. and Roberts, G. O. (2003). [slice sampling]: Discussion. *The Annals of Statistics*, 31(3):748–753.

Mira, A. and Tierney, L. (2002). Efficiency and convergence properties of slice samplers. *Scandinavian Journal of Statistics*, 29:1–12(12).

Neal, R. M. (1997). Markov chain monte carlo methods based on 'slicing' the density function. Technical report, Department of Statistics, University of Toronto.

Neal, R. M. (2003a). Slice sampling. *The Annals of Statistics*, 31(3):705–741.

Neal, R. M. (2003b). [slice sampling]: Rejoinder. *The Annals of Statistics*, 31(3):758–767.

Nott, D. J. and Leonte, D. (2004). Sampling schemes for bayesian variable selection in generalized linear models. *Journal of Computational and Graphical Statistics*, 13(2):362–382.

Roberts, G. O. and Rosenthal, J. S. (1999). Convergence of slice sampler markov chains. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 61:643–660(18).

Roberts, G. O. and Rosenthal, J. S. (2002). The polar slice sampler. *Stochastic Models*, 18(2):257–280.

Roberts, G. O. and Sahu, S. K. (1997). Updating schemes, correlation structure, blocking and parameterization for the gibbs sampler. *Journal of the Royal Statistical Society: Series B (Methodological)*, 59(2):291–317.

Shahbaba, B. and Neal, R. (2006). Gene function classification using bayesian models with hierarchy-based priors. *BMC Bioinformatics*, 7(1):448.

Sun, S., Greenwood, C. M., and Neal, R. M. (2007). Haplotype inference using a bayesian hidden markov model. *Genetic Epidemiology*, 31(8):937–948.

Whiley, M. and Wilson, S. P. (2004). Parallel algorithms for markov chain monte carlo methods in latent spatial gaussian models. *Statistics and Computing*, 14(3):171–179.

Yan, J., Cowles, M. K., Wang, S., and Armstrong, M. P. (2007). Parallelizing MCMC for Bayesian spatiotemporal geostatistical models. *Statistics and Computing*, 17(4):323–335.