Chapter 18

Parallel Bayesian Computation

Darren J Wilkinson, *University of Newcastle, UK*
d.j.wilkinson@ncl.ac.uk

18.1 Introduction

The use of Bayesian inference for the analysis of complex statistical models has increased dramatically in recent years, in part due to the increasing availability of computing power. There are a range of techniques available for carrying out Bayesian inference, but the lack of analytic tractability for the vast majority of models of interest means that most of the techniques are numeric, and many are computationally demanding. Indeed, for high-dimensional non-linear models, the only practical methods for analysis are based on Markov chain Monte Carlo (MCMC) techniques, and these are notoriously compute intensive, with some analyses requiring weeks of CPU time on powerful computers. It is clear therefore that the use of parallel computing technology in the context of Bayesian computation is of great interest to many who analyse complex models using Bayesian techniques.

Section 18.2 considers the key elements of Bayesian inference, and the notion of graphical representation of the conditional independence structure underlying a statistical model. This turns out to be key to exploiting partitioning of computation in a parallel environment. Section 18.3 looks at the issues surrounding Monte Carlo simulation techniques in a parallel environment, laying the foundations for the examination of parallel MCMC in Section 18.4. Standard pseudo-random number generators are not suitable for use in a parallel setting, so this section examines the underlying reasons and the solution to the problem provided by parallel pseudo-random number generators.

Parallel MCMC is the topic of Section 18.4. There are two essentially different strategies which can be used for parallelising an MCMC scheme (though these may be combined in a variety of ways). One is based on running multiple MCMC chains in parallel and the other is based on parallelisation of a single MCMC chain. There are different issues related to the different strategies, and each is appropriate in different situations. Indeed, since MCMC in complex models is somewhat of an art-form anyway, with a range of different possible algorithms and trade-offs even in the context of a non-parallel computing environment, the use of a parallel computer adds an additional layer of complexity to the MCMC algorithm design process. That is, the trade-offs one would adopt for the design of an efficient MCMC algorithm for the analysis of a given statistical algorithm in a non-parallel environment may

well be different in a parallel setting, and could depend on the precise nature of the available parallel computing environment. Section 18.4 will attempt to give some background to the kind of trade-offs that are made in the design of an efficient MCMC algorithm, and into the reasons why one might design an algorithm differently depending on the available hardware and software.

A wide range of parallel hardware is now available. Dedicated multi-processor parallel computers offer outstanding inter-processor communication speeds, but networked collections of fairly ordinary PCs running a Unix-derived operating system such as Linux offer the opportunity to get super-computer performance for relatively little capital outlay. Linux "Beowulf" clusters are rapidly becoming the *de facto* standard for affordable super-computing. The examples given in the later sections use a small 8-CPU Linux cluster. The cluster was formed by four dual CPU PCs together with a cheap file server linked via switched 100 Mbit network (funded by a small grant from the Royal Society). However, the principles apply also to dedicated parallel computers and to much larger Beowulf clusters using faster (Gigabit or ATM) networking infra-structure. As already mentioned, the design of algorithms will often depend to some extent on the parallel architecture. In particular, the number of processors and the speed (and latency) of inter-processor communication are of particular importance. The examples assume a homogeneous cluster (ie. all processors run at the same speed linked by a fairly uniform networking infrastructure), but are easily adapted to non-homogeneous environments.

Parallel computing relies on the ability to write programs which exploit multiple processors and processes and transfer information between them. Rather than developing communication protocols from scratch using low level network libraries, it is far more efficient to use specially written message-passing libraries specifically designed for the development of parallel applications. The Parallel Virtual Machines (PVM) library and the Message Passing Interface (MPI) are the most popular choices. MPI is a newer more sophisticated library, and is probably a better choice for people new to parallel computing without an existing PVM code base. The small number of MPI calls necessary for the development of parallel Bayesian inference algorithms will be introduced and illustrated as they are required.

Where necessary, ideas are illustrated with the description of algorithms or simple pieces of code written in ANSI `'C'`. Experience of programming will be assumed, but no experience of parallel computing is required, nor is extensive knowledge of ANSI `'C'`. Note that the examples adapt easily to other languages, and hence are not ANSI `'C'` specific. In particular, the key software libraries used (MPI and SPRNG) both have Fortran interfaces, and hence can be used just as easily from Fortran programs.

18.2 Bayesian inference

18.2.1 Introduction

This section introduces the key concepts necessary for the development of Bayesian inference algorithms. It also defines the notation used in the subsequent sections. For an introduction to Bayesian inference, O'Hagan (1994) covers the theory, and Gamerman (1997) provides a good introduction to modern computational techniques based on MCMC. In the simplest continuous setting we are interested in inference for the parameter vector ϕ of a probability (density) model $p(y|\phi)$ giving rise to an observed data vector y. If we treat the parameters as uncertain, and allocate to them a "prior" probability density $\pi(\phi)$, then Bayes' theorem

gives the "posterior" density

$$\pi(\phi|y) = \frac{\pi(\phi)p(y|\phi)}{p(y)},$$

where $p(y)$ is the marginal density for y obtained by integrating over the prior for ϕ. Since $\pi(\phi|y)$ is regarded as a function of ϕ for fixed (observed) y, we can re-write this as

$$\pi(\phi|y) \propto \pi(\phi)p(y|\phi),$$

so that the posterior is proportional to the prior times the model likelihood function.

For more complex models this simple description hides some important details. If y is the observed data then ϕ represents everything else in the model that is not directly observed. This will no doubt include "conventional" model parameters, but may well include other aspects of a model, such as "nuisance" parameters, random effects and/or missing data. In the context of more complex models it is therefore often helpful to partition $\phi = (\sigma, \theta)$, where σ represents "conventional" model parameters of direct inferential interest and θ represents all other details of the model which are not directly observed. Indeed, it is the ability of the Bayesian framework to easily handle models of this form, by treating all aspects of a model in a unified way that makes it so attractive to a broad range of researchers. The prior is usually factored as $\pi(\phi) = \pi(\sigma, \theta) = \pi(\sigma)\pi(\theta|\sigma)$, so that Bayes' theorem now becomes

$$\pi(\sigma, \theta|y) \propto \pi(\sigma)\pi(\theta|\sigma)p(y|\sigma, \theta) \tag{18.1}$$

where the constant of proportionality is independent of both σ and θ. If σ is of primary concern, then real interest will be in the marginal posterior obtained by integrating θ out of (18.1), that is

$$\pi(\sigma|y) \propto \pi(\sigma) \int \pi(\theta|\sigma)p(y|\sigma, \theta)d\theta$$

which can be written as

$$\pi(\sigma|y) \propto \pi(\sigma)p(y|\sigma), \tag{18.2}$$

where

$$p(y|\sigma) = \int \pi(\theta|\sigma)p(y|\sigma, \theta)d\theta \tag{18.3}$$

is the marginal likelihood for θ given the data, and will typically be either unavailable or computationally expensive to evaluate. Before moving on, however, it is worth noting that there is an alternative representation of (18.3) that follows immediately from Bayes theorem:

$$p(y|\sigma) = \frac{\pi(\theta|\sigma)p(y|\sigma, \theta)}{\pi(\theta|\sigma, y)}. \tag{18.4}$$

Chib (1995) refers to this as the basic marginal likelihood identity (BMI), and it is sometimes tractable in situations where it is not at all clear how to tackle the integral in (18.3) directly.

The computational difficulties in Bayesian inference arise from the intractability of high-dimensional integrals such as the constants of integration in (18.1) and (18.2). These are typically not only analytically intractable but also difficult to obtain numerically. Although the constant of proportionality in (18.2) appears easier to evaluate, the difficult integration problem has simply been pushed to the evaluation of the marginal likelihood (18.3). Even where the constants of proportionality are known, there are further integration problems associated with obtaining the marginal distributions of various functions of the parameters of interest, as well as computing expectations of summary statistics.

In some situations it will be helpful to further partition $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_p)$ and/or $\theta = (\theta_1, \theta_2, \ldots, \theta_q)$, but there is no suggestion that the sub-components are univariate unless explicitly stated.

Example 1 *In order to make things concrete, consider the simple one-way ANOVA model with random effects:*

$$y_{ij} = \mu + \alpha_i + \varepsilon_{ij}, \quad i = 1, 2, \ldots, n, \quad j = 1, 2, \ldots, r_i \tag{18.5}$$

where

$$\alpha_i \sim N(0, \sigma_\alpha^2) \quad \textit{and} \quad \varepsilon_{ij} \sim N(0, \sigma_\varepsilon^2).$$

In fact the random variables ε_{ij} are redundant, and should be removed from the specification by re-writing (18.5) as

$$y_{ij} \sim N(\mu + \alpha_i, \sigma_\varepsilon^2).$$

Here, one generally regards the α_i to be "missing data", and so the variables would be partitioned as $\sigma = (\mu, \sigma_\alpha, \sigma_\varepsilon)$, $\theta = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ and y as the vector of all y_{ij}. There are other ways of partitioning these parameters, however, depending on the variables of interest and the computational techniques being adopted. For example, the variable μ could be moved from σ to θ if interest is mainly concerned with the variance components σ_α and σ_ε and linear Gaussian techniques are to be used to understand the rest of the model conditional on σ. With the partitioning as initially described, we know that

$$\pi(\theta|\sigma) = \prod_{i=1}^{n} N(\alpha_i; 0, \sigma_\alpha^2) \quad \textit{and} \quad \pi(y|\theta, \sigma) = \prod_{i=1}^{n} \prod_{j=1}^{r_i} N(y_{ij}; \mu + \alpha_i, \sigma_\varepsilon),$$

where

$$N(x; \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{ -\frac{1}{2} \left(\frac{x - \mu}{\sigma} \right)^2 \right\}.$$

Note that $\pi(\sigma)$ has not yet been specified, but that this must be done before any Bayesian analysis can be conducted.

18.2.2 Graphical models and conditional independence

In order to exploit a parallel computing environment there must be a way of decomposing the given computational task into parts which may be carried out independently of one another. In the context of complex statistical models the notion of conditional independence of model components provides a natural way of breaking down the model into manageable pieces. Conditional independence structures have a natural representation in terms of graphs, and hence models characterised by their conditional independence structure are often referred to as graphical models.

For a basic introduction to graphical models see Whittaker (1990) and for a complete account, see Lauritzen (1996). Here we introduce the essential concepts and notation required for later sections. Consider a finite directed graph $\mathcal{G} = (V, E)$ with vertex set $V = \{X_1, X_2, \ldots, X_n\}$ and finite edge set $E = \{(v, v') | v, v' \in V, v \to v'\}$. The graph is acyclic if there is no directed sequence of edges starting and finishing at the same vertex. A directed acyclic graph is known as a DAG. The *parents* of $v \in V$ are given by

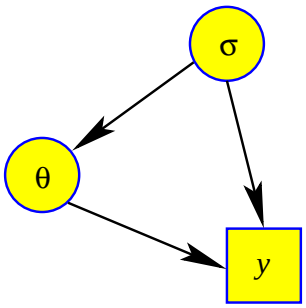

Figure 18.1: DAG representing the basic model factorisation

$\text{pa}(v) = \{v' \in V | (v', v) \in E\}$. The density of a random vector $X = (X_1, X_2, \ldots, X_n)$ is said to factor according to $\mathcal{G}$ if the probability density for X can be written as

$$\pi(x) = \prod_{i=1}^{n} \pi(x_i | \text{pa}(x_i)). \tag{18.6}$$

It follows that X_i is conditionally independent of every node which is not a direct descendent given only its parents. Thus for any X_j which is not a descendent of X_i we write $X_i \perp\!\!\!\perp X_j | \text{pa}(X_i)$.

Example 2 *The basic model factorisation*

$$\pi(\sigma, \theta, y) = \pi(\sigma)\pi(\theta|\sigma)p(y|\sigma, \theta)$$

has the DAG representation illustrated in Figure 18.1. Note that there are no (non-trivial) conditional independencies implied by this DAG, corresponding to the fact that there are no restrictions placed on the form of the joint density imposed by the chosen factorisation. Note also that we could have chosen to factorise the joint density in a different order, and hence there is typically more than one valid DAG corresponding to any given conditional independence structure.

Undirected graphs can also be used to represent conditional independence structures. These graphs have the property that each node is conditionally independent of every other node given only its *neighbours*, defined by $\text{ne}(v) = \{v' \in V | v' \sim v\}$ where $\sim$ here denotes adjacency in the graph. Thus for every X_j not directly joined to X_i we write $X_i \perp\!\!\!\perp X_j | \text{ne}(X_i)$. Such graphs have other important conditional independence properties (and also have implications for the factorisation of the joint density) but see Lauritzen (1996) for details. It is worth noting that a valid undirected conditional independence graph can be obtained from a DAG by first "marrying" all pairs of parents of each node, and then dropping "arrows". An undirected graph formed in this way is often referred to as a *moral* graph.

18.2.3 Local computation in graphical models

Certain classes of statistical model are computationally tractable, at least in principle, but become problematic in higher dimensions. For problems which are purely discrete, for example, Bayesian inference is simply a matter of constructing conditional and marginal probability tables and forming weighted sums of probabilities to obtain posterior probabilities of

interest. However, in practice the probability tables involved quickly become infeasibly large and so a direct approach to the problem fails. Fortunately, if the problem has a relatively "sparse" conditional independence structure, represented by a DAG without too many edges, then the conditional independence structure may be exploited so that calculations and computations done at any time involve only "nearby" DAG nodes, yet the results of these "local" computations may be combined to obtain "global" posterior probabilities of inferential interest. Essentially, the DAG is first *moralised* into an undirected graph and then coerced into a tree-shaped conditional independence graph known as a *clique-tree* or *junction tree*. Computations (which involve only adjacent nodes of the tree) are then carried out at each node and then passed on to neighbours in order to solve a given global problem. Pearl (1988) gives an excellent introduction to this area, and a more detailed treatment of the problem is considered in Cowell et al. (1999), to which the reader is referred for further information.

There are other classes of models for which local computation is possible. The class of fully specified linear Gaussian models is of particular importance in applied statistics. Again, the conditional independence structure can be exploited to carry out calculations of interest in Bayesian inference based on local information. Wilkinson and Yeung (2002) study this problem from the perspective of simulating unobserved variables in the model conditional on all observations. It turns out that exact local computation techniques are also possible for a certain class of models known as conditionally Gaussian (CG) models which involve a mixture of discrete and linear Gaussian random variables; see Lauritzen (1992) and Cowell et al. (1999) for further details.

For pure linear Gaussian systems there is another way of tackling the problem which utilises the fact that the joint distribution of all variables in the problem is multivariate normal, and that although the variance matrix is typically very large, its inverse (the "precision" matrix) is tractable and very sparse, having non-zero elements only where there are arcs in the associated conditional independence graph. Consequently, computations can be conducted by working with a sparse matrix representation of the problem and applying sparse matrix algorithms to the resulting computational problems in order to automatically exploit the sparseness and locality of the conditional independence structure. Wilkinson and Yeung (2003) describe this approach in detail and show how to apply it in practice, describing a free software library, `GDAGsim`, which implements the techniques. If, rather than being DAG based, the linear Gaussian system is derived from a Markov random field, the techniques of Rue (2001), implemented in the library `GMRFlib`, can be used instead.

18.2.4 Parallel methods for local computation

Even using an ordinary serial computer, local computation methods allow the exact analysis of problems that would be difficult to contemplate otherwise. However, the resulting algorithms are often computationally expensive, making exploitation of high performance parallel computers attractive. It turns out that this is relatively straightforward if the number of available processors is quite small, but the algorithms do not scale well as the number of processors increases. In the context of sending messages around a tree shaped graph, messages in different branches of the tree can be computed and passed in parallel, allowing a modest speedup if the "diameter" of the graph is relatively small relative the the total number of nodes. The issues of parallel message-passing and "scheduling of flows" are considered in some detail in Pearl (1988) and Cowell et al. (1999).

Similar considerations apply to the sparse matrix methods for linear Gaussian systems.

Parallel direct sparse matrix algorithms such as provided by the PSPASES library or ScaLAPACK can be used in place of conventional serial sparse matrix algorithms; however, such algorithms tend to scale poorly to large numbers of processors. For certain kinds of computation, parallel *iterative* solvers may be used. These scale much better than direct methods, but cannot be used for all computations of interest. See George and Liu (1981) and Kontoghiorghes (2000) for further details of the range of techniques that are useful in this context.

18.3 Monte Carlo simulation

18.3.1 Introduction

Despite the obvious utility of the exact methods discussed in Section 18.2.3, they typically fail to provide a complete solution to the problem of Bayesian inference for the large complex non-linear statistical models that many statistical researchers are currently concerned with. In such situations, one is faced with the problem of high-dimensional numerical integration, and this is a classically hard problem. If the dimensions involved are relatively low, then it may be possible to use standard numerical quadrature techniques. One advantage of such techniques is that they are "embarrassingly" parallel, in that the integration space can be easily partitioned and each processor can integrate the space allocated to it and then the results can be combined in order to give the final result. This problem is considered in almost every introduction to parallel computing; see for example, Pacheco (1997), for further details.

Once the dimensions involved are large (say, more than 10), then standard quadrature methods become problematic, even using large parallel computers. However, many statistical problems of interest involve integrating over hundreds or even thousands of dimensions. Clearly a different strategy is required. One way to tackle high-dimensional integrals is to use Monte Carlo simulation methods. Suppose interest lies in the integral

$$I = \mathrm{E}_\Pi(t(\phi)) = \int t(\phi) d\Pi(\phi) = \int t(\phi)\pi(\phi)d\phi$$

for some high-dimensional ϕ. If n values of ϕ can be sampled independently from the density $\pi(\phi)$ then I may be approximated by

$$I_1 = \frac{1}{n}\sum_{i=1}^{n} t(\phi^{(i)}).$$

Even if sampling directly from $\pi(\phi)$ is problematic, suppose it is possible to independently sample values of ϕ from a density $f(\phi)$ which has the same support as $\pi(\phi)$, then I may be re-written

$$I = \int \frac{t(\phi)\pi(\phi)}{f(\phi)} f(\phi)d\phi = \int \frac{t(\phi)\pi(\phi)}{f(\phi)} dF(\phi) = \mathrm{E}_F\left(\frac{t(\phi)\pi(\phi)}{f(\phi)}\right).$$

Consequently, given n samples from $f(\phi)$, I may be approximated by

$$I_2 = \frac{1}{n}\sum_{i=1}^{n} \frac{t(\phi^{(i)})\pi(\phi^{(i)})}{f(\phi^{(i)})}.$$

The law of large numbers (LLN) assures us that both I_1 and I_2 are strongly consistent unbiased estimators of I, with I_1 preferred to I_2 if available. The quality (precision) of I_2 will depend strongly on how similar $f(\phi)$ is to $\pi(\phi)$. The problem with Monte Carlo methods is that the errors associated with the estimates are proportional to $1/\sqrt{n}$, so very large samples are required for good estimation. See Ripley (1987) for further details of stochastic simulation and Monte Carlo methods, including variance reduction techniques.

Monte Carlo techniques are very computationally intensive, but are ideal in the context of parallel computing. For simplicity, consider the evaluation of I_1. Since the samples are independent, it is possible to divide up the n required samples between the available processors. Each processor can then generate and summarise its samples, and once all processors have finished, the summaries can be summarised to give the final result. In a homogeneous environment, it is probably desirable simply to divide up the required number of samples equally between the processors, but in an inhomogeneous environment that is most likely not a good strategy. Rosenthal (2000) examines strategies for parallel Monte Carlo in a non-homogeneous environment, considering a range of issues including time-limited resources and unreliable processors.

Consider now the problem of evaluating I_1 using N processors where $n = Nm$, for some integer m. The algorithm for parallel simulation can be described as follows.

1. Master program computes $m = n/N$, and passes m to each available processor
2. Each processor (k):
   (a) simulates m independent realisations of ϕ
   (b) computes $S_k = \sum_{i=1}^{m} t(\phi^{(i)})$
   (c) passes S_k back to master program
3. Master program collects and sums the S_k to give final sum S.
4. Master program returns S/n

Section 18.3.6 describes a complete computer program to implement this algorithm, but relies on some topics not yet covered.

18.3.2 Pseudo-random number generation (PRNG)

The above algorithm gives the essential logic, but there are some issues relating to simulation of random quantities on a computer which need to be addressed. The algorithm clearly assumes that all simulated realisations of the random quantity ϕ are independent. This requires not only that the simulated realisations generated on a given processor are independent of one another, but also that the simulated values on different processors are independent of one another. This causes some difficulties with conventional pseudo-random number generation algorithms based on a single underlying "stream". The pseudo-random number generators used by almost all computers are in fact deterministic algorithms carefully designed to give a sequence of numbers which give every appearance of being both uniformly distributed and independent of one another. These can then be transformed to give independent realisations of other distributions of interest. Standard pseudo-random number generators are unsatisfactory in a parallel context, but to see why, one must understand a little about how they work.

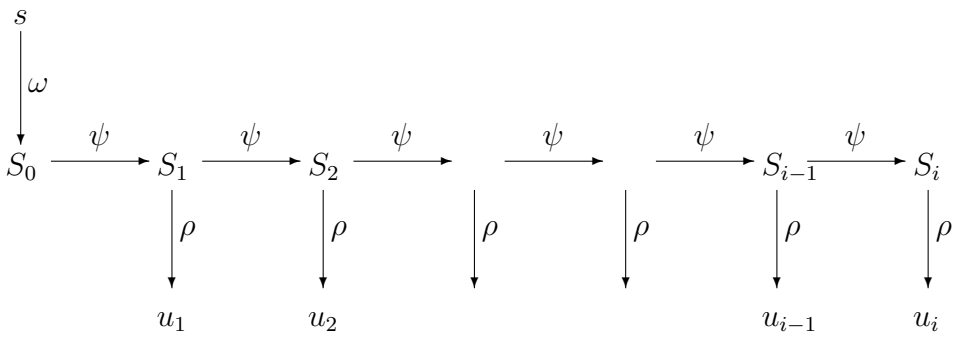

Figure 18.2: Illustration of the sequence of deterministic mappings involved in the generation of a pseudo-random number stream

Pseudo-random number generators start with a seed, s. There may well be a default seed, or in some situations a seed may be chosen automatically based on things such as the computer system clock, but s is always present, and for most random generators, may be explicitly set by the user. The heart of the random number generator is its internal state, S, which is finite, and maintained throughout the time that the generator is used by the program. The initial state is set according to $S_0 = \omega(s)$, for some deterministic function $\omega(\cdot)$. When a uniform random number is required, the state is first updated via $S_i = \psi(S_{i-1})$ and then a uniform random number is computed as $u_i = \rho(S_i)$ for some deterministic functions $\psi(\cdot)$ and $\rho(\cdot)$ (Figure 18.2). Clearly the most important part of the generator is the function $\psi(\cdot)$, which although deterministic, is carefully chosen to give an apparently random sequence of states. The way that $\psi(\cdot)$ is constructed is not of particular concern here, but the interested reader is referred to Ripley (1987) for an introduction. Note that the resulting uniform random variate is likely to be further deterministically transformed in order to give a sample from some non-uniform distribution of interest (Devroye 1986).

18.3.3 PRNG using the GNU Scientific Library (GSL)

The GNU Scientific Library (GSL) contains an extensive set of functions for (serial) pseudo-random number generation and transformation. It is perhaps useful to introduce some of these functions at this point. Below is a complete ANSI `'C'` program to simulate 10 $U(0,1)$ random variates and print them out to the console.

```
1  #include <gsl/gsl_rng.h>

2  int main(void)
3  {
4    int i; double u; gsl_rng *r;
5    r = gsl_rng_alloc(gsl_rng_mt19937);
6    gsl_rng_set(r,0);
7    for (i=0;i<10;i++) {
8      u = gsl_rng_uniform(r);
9      printf("u(%d) = %f\n",i,u);
10   }
```

```
11    exit(EXIT_SUCCESS);
12  }
```

Note that the line numbers are not part of the program. The above program will compile with a command like:

```
gcc gsl_rng_demo.c -o gsl_rng_demo -lgsl -lgslcblas
```

The key lines of this program are as follows. Line 1 makes the random number generator functions of the GSL library available to the program. Line 4 allocates the variables used by the program, including a variable `r`, which contains information about the random number stream being used, including its internal state. Line 5 initialises the generator, setting it to be based on a "Mersenne Twister" generator with a period of $2^{19937} - 1$. Line 6 sets the seed of the generator to the (default) value of 0. Line 8 requests the next uniform random variate from the stream. Due to the way that the streams are handled via a variable (here `r`), it is possible to use several streams (possibly of different types) within a single (serial) program.

The GSL has many different kinds of uniform generator, as well as an extensive range of functions for non-uniform random number generation. Importantly, the library is designed so that any kind of uniform stream can be used in conjunction with any non-uniform generator function. Consider the complete program below for simulating standard normal deviates.

```
 1  #include <gsl/gsl_rng.h>
 2  #include <gsl/gsl_randist.h>

 3  int main(void)
 4  {
 5    int i; double z; gsl_rng *r;
 6    r = gsl_rng_alloc(gsl_rng_mt19937);
 7    gsl_rng_set(r,0);
 8    for (i=0;i<10;i++) {
 9      z = gsl_ran_gaussian(r,1.0);
10      printf("z(%d) = %f\n",i,z);
11    }
12    exit(EXIT_SUCCESS);
13  }
```

Line 2 makes the non-uniform random number generation functions available to the program, and line 9 requests the generation of a standard normal random variable using the given stream `r` as a source of randomness. In this case, the uniform stream is based on a "Mersenne Twister", but the `gsl_ran_*` functions will all happily use any other GSL random number stream that is supplied.

Random number streams of the above form make perfect sense in the context of a serial program, where random variates are read sequentially from a single "stream". However, in a parallel environment it is not at all convenient for all processors to share such a single stream — it involves a prohibitive amount of communication overhead. Consequently, it is highly desirable for each processor to maintain its own stream (state), but this turns out to be problematic.

Clearly if each processor maintains its own random number stream then the random variates generated on that processor will appear to be independent of one another (assuming the generator is good). However, we also require that the variates are independent *across*

processors, and this is where the problems lie. Clearly if each processor starts with the *same seed* then the random number streams on each processor will be identical, and each processor will produce exactly the same computations. In this case the results are very clearly not independent across processors. A widely used "solution" to this problem is to use *different seeds* on each processor, but this too presents some difficulties. It is important to realise that the internal state of a pseudo-random number generator is necessarily finite, as it is stored on a digital computer. Given the deterministic nature of the state updating function $\psi(\cdot)$, it follows that the generator will *cycle* through a fixed set of states in a fixed order at some point. The length of cycle is known as the *period* of the generator, and it is obviously desirable for the period to be larger than the number of random variates that will be required for the simulation problem under consideration (with good modern generators this is rarely a problem). However, suppose that a program is running in a parallel environment and that seed s^A is used on processor A and seed s^B is used on processor B. If one considers the initial states of the random number generators on processors A and B these are $S_0^A = \omega(s^A)$ and $S_0^B = \omega(s^B)$. Now let $\mathcal{S}^A$ denote the *orbit* of S_0^A; that is $\mathcal{S}^A = \{\psi^n(S_0^A)|n \in \mathbb{N}\}$, where $\psi^n(\cdot)$ represents the n-fold composition of the map $\psi(\cdot)$. If it is the case that $S_0^B \in \mathcal{S}^A$ (as is typically the case for random number generators designed with serial computers in mind), then it is clear that for sufficiently long simulation runs the streams on processors A and B will overlap. It follows that unless the seeding algorithms used are extremely sophisticated, then for any substantial simulation there is a non-negligible probability that the random number streams on parallel processors will have some overlap. This leads to non-independence of the streams across processors and is obviously undesirable.

18.3.4 Parallel pseudo-random number generation (PPRNG)

In order to overcome the difficulties with the use of serial pseudo-random number streams in a parallel environment, a theory of parallel pseudo-random number generation has been developed (Srinivasan et al. 1999). There are several different strategies one can employ for adapting standard serial generators into the parallel context, and a detailed examination would not be appropriate here. However, it is useful to have some idea how such generators work, and hence a brief description will now be given of the key idea behind an important class of PPRNG algorithms that are said to be *initial value parameterised*.

Using the notation from Section 18.3.2, the main novelty in this class of PPRNG is the use of an initialisation function $\omega(\cdot)$ which depends on the processor number. So for each processor $k = 1, 2, \ldots, N$ we map a single *global* seed, s to N different initial states given by $S_0^k = \omega^k(s)$. Then for each k, a stream is generated in the usual way using $S_i^k = \psi(S_{i-1}^k)$ and $u_i^k = \rho(S_i^k)$ (Figure 18.3). The key to independence of the resulting streams is in the careful choice of updating function $\psi(\cdot)$ and initialisation functions $\omega^k(\cdot)$ which ensure that the N resulting orbits, $\mathcal{S}^1, \mathcal{S}^2, \ldots \mathcal{S}^N$ are non-overlapping and the same known size irrespective of the choice of global seed s. It is helpful to regard the state space of the PPRNG as forming a toroidal lattice with rows of the lattice forming the streams obtainable on a single processor and the initialisation functions $\omega^k(\cdot)$ providing a mechanism for mapping an arbitrary seed onto a given row of the lattice.

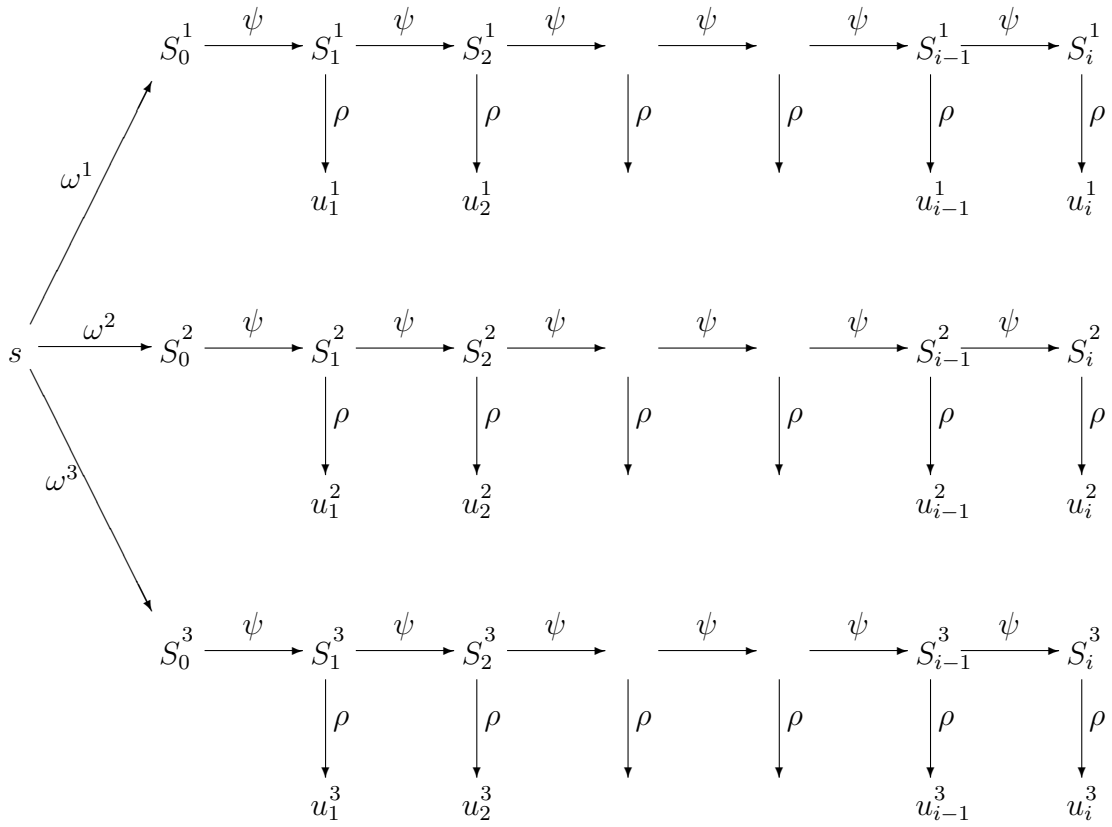

Figure 18.3: Illustration of the deterministic mappings involved in the generation of an independent family of pseudo-random number streams based on parameterisation of initial values

18.3.5 The scalable parallel random number generator (SPRNG)

Mascagni and Srinivasan (2000) describe a useful family of parameterised parallel random number generators implemented in a freely available software library, SPRNG. Being parallel, the library relies on some form of parallel communication (simply to number the different streams on each processor correctly), and the authors of this library have ensured that it is easy to use in conjunction with the Message-Passing Interface (MPI); see Pacheco (1997) for an introduction to MPI, and Snir et al. (1998) for reference. The SPRNG library has a number of sophisticated features, including the ability to use multiple streams on each processor. However, for most Monte Carlo (and Markov chain Monte Carlo) applications, a single independent stream on each processor is sufficient, and for this there is a simple interface which is very easy to use. Consider the following program which generates 10 $U(0,1)$ random variates on each available processor.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <mpi.h>
4 #define  SIMPLE_SPRNG
5 #define  USE_MPI
```

```
 6 #include "sprng.h"

 7  int main(int argc,char *argv[])
 8  {
 9    double rn; int i,k;
10    MPI_Init(&argc,&argv);
11    MPI_Comm_rank(MPI_COMM_WORLD,&k);
12    init_sprng(DEFAULT_RNG_TYPE,0,SPRNG_DEFAULT);
13    for (i=0;i<10;i++) {
14      rn = sprng();
15      printf("Process %d, random number %d: %f\n", k, i+1, rn);
16    }
17    MPI_Finalize();
18    exit(EXIT_SUCCESS);
19  }
```

This program can be compiled with a command like

```
mpicc sprng_demo.c -o sprng_demo -lsprng
```

and run (on 8 processors) with a command like

```
mpirun -np 8 sprng_demo
```

This (and most) MPI programs use the single-program multiple-data (SPMD) parallel programming paradigm. Exactly the same program is run on every available processor. However, since each process is aware of its "process number", different processes are able to behave differently by using standard conditional directives.

Going through the above code in detail, line 3 makes the MPI functions available to the program. Lines 4 and 5 declare that we are using the "simple" interface to SPRNG and that we are running SPRNG on top of MPI. Line 6 makes the SPRNG library functions available to the program. Line 10 initialises MPI, making sure that each copy of the process is started correctly and initialised with any command line arguments. Line 11 records the process number in a variable `k`, so that it can be used by the program. Line 12 initialises the SPRNG random number generator (on each process), with a common global seed of 0. Line 14 generates a $U(0,1)$ variable using SPRNG, and line 17 is used to ensure that the MPI program terminates correctly.

Now as previously mentioned, in the context of Monte Carlo integration the uniform random variables generated by SPRNG will typically require transforming to quantities from some non-uniform distribution of interest. The GSL functions of the form `gsl_ran_*` do exactly this, but only operate on random number streams of type `gsl_rng`. The author has written a small piece of code, `gsl-sprng.h` (available from his web site), which wraps up the simple SPRNG generator as a GSL PRNG (`gsl_rng_sprng20`) enabling the development of parallel non-uniform PRNG codes. The following program generates 10 $Pois(2)$ random quantities on each available processor.

```
1  #include <mpi.h>
2  #include <gsl/gsl_rng.h>
3  #include "gsl-sprng.h"
4  #include <gsl/gsl_randist.h>
```

```
 5  int main(int argc,char *argv[])
 6  {
 7    int i,k,po; gsl_rng *r;
 8    MPI_Init(&argc,&argv);
 9    MPI_Comm_rank(MPI_COMM_WORLD,&k);
10    r=gsl_rng_alloc(gsl_rng_sprng20);
11    for (i=0;i<10;i++) {
12      po = gsl_ran_poisson(r,2.0);
13      printf("Process %d, random number %d: %d\n", k, i+1, po);
14    }
15    MPI_Finalize();
16    exit(EXIT_SUCCESS);
17  }
```

This combination of ANSI 'C', MPI, SPRNG and the GSL seems currently to be the best framework for the development of sophisticated parallel Monte Carlo (and Markov chain Monte Carlo) algorithms.

18.3.6 A simple parallel program for a Monte Carlo integral

This section will finish with a simple example Monte Carlo integral. Suppose we require

$$I = \int_0^1 \exp\{-u^2\}du = \mathrm{E}_U\big(\exp\{-U^2\}\big) \simeq \frac{1}{n}\sum_{i=1}^{n}\exp\{-u_i^2\},$$

where $U \sim U(0,1)$ and $u_i,\ i = 1, 2, \ldots, n$ are independent samples from this distribution. The following program simulates $10,000$ variates on each processor, calculates the partial sums, then "gathers" the N partial sums back together on process number 0 where the final result is computed and printed.

```
 1  #include <math.h>
 2  #include <mpi.h>
 3  #include <gsl/gsl_rng.h>
 4  #include "gsl-sprng.h"

 5  int main(int argc,char *argv[])
 6  {
 7    int i,k,N; double u,ksum,Nsum; gsl_rng *r;
 8    MPI_Init(&argc,&argv);
 9    MPI_Comm_size(MPI_COMM_WORLD,&N);
10    MPI_Comm_rank(MPI_COMM_WORLD,&k);
11    r=gsl_rng_alloc(gsl_rng_sprng20);
12    for (i=0;i<10000;i++) {
13      u = gsl_rng_uniform(r);
14      ksum += exp(-u*u);
15    }
16    MPI_Reduce(&ksum,&Nsum,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
17    if (k == 0) {
18      printf("Monte carlo estimate is %f\n", (Nsum/10000)/N );
```

```
19    }
20    MPI_Finalize();
21    exit(EXIT_SUCCESS);
22  }
```

The key new features of this program are the use of `MPI_Comm_size` (line 9) to obtain the number of processors, and the use of `MPI_Reduce` (line 16) to add together the N partial sums and store the result on process 0. Running the program should give a result close to the true answer of 0.74682.

18.4 Markov chain Monte Carlo (MCMC)

18.4.1 Introduction

For computations involving high-dimensional posterior distributions, it is usually impractical to implement pure Monte Carlo solutions. Instead, the most effective strategy is often to construct a Markov chain with equilibrium distribution equal to the posterior distribution of interest. There are a variety of strategies which can be used to achieve this; see Brooks (1998) and Gamerman (1997) for a good introduction. From a practical perspective, the chain is initialised at an essentially arbitrary starting state, and successive values are sampled based on the previous value using a PRNG. In principle, after a "burn-in" period the samples should be from the equilibrium distribution, and these can be studied in order to gain insight into the posterior of interest.

Even in the context of a single serial computer, there is some controversy surrounding the question of whether or not it is better to run one long chain or several shorter ones (Gelman and Rubin 1992; Geyer 1992). The argument in favour of one long run is that the burn-in only happens once, whereas for several shorter chains, each must burn-in, resulting in many wasted samples. The arguments in favour of several chains concern the possibility of being able to better diagnose convergence to the equilibrium distribution. In the context of a serial computer the author tends to side with the one long run camp, but in a parallel environment the situation is rather different. Provided that burn-in times are relatively short, running a chain on each processor can often be the most effective way of utilising the available hardware.

18.4.2 Parallel chains

It is important to bear in mind that for any given posterior distribution, $\pi(\phi|y)$, there are many different MCMC algorithms that can be implemented, all of which have $\pi(\phi|y)$ as the unique equilibrium distribution. When designing an MCMC algorithm for a particular problem there are a range of trade-offs that are made. Often the easiest scheme to implement is based on a Gibbs sampler or simple Metropolis-within-Gibbs scheme which cycles through a large number of low-dimensional components. Although easy to implement, such schemes often have very poor mixing and convergence properties if the dimension of ϕ is high. The output of such a scheme generally has a significant proportion of the run discarded as burn-in, and the iterations left over are often "thinned" (keeping, say, only 1 in 100 iterations) in order to reduce dependence in the samples used for analysis. The issue of burn-in is of particular concern in a parallel computing environment. The fact that every processor must spend

a significant proportion of its time producing samples that will be discarded places serious limitations on the way the performance of the algorithm scales with increasing numbers of processors (Rosenthal 2000). Given that there are a variety of ways of improving the mixing of MCMC algorithms, including blocking and reparameterisation (Roberts and Sahu 1997), it may be particularly worth while spending some time improving the mixing of the sampler if a parallel chains strategy is to be adopted (Section 18.4.3). If this is not practical, then parallelisation of a single MCMC chain is possible, but this too presents difficulties, and is unlikely to scale well to very large numbers of processors in general (Section 18.4.4).

Whenever it is felt to be appropriate to use a parallel chain approach, producing code to implement it is extremely straightforward. In this case it is often most straightforward to first develop the code for a single processor, and then convert to a parallel program once it is debugged. Debugging MCMC code and parallel code are both devilishly difficult, and so it is best to try to avoid doing the two simultaneously if at all possible!

Example 3 *Consider generation of a standard normal random quantity using a random walk Metropolis-Hastings sampler with* $U(-\alpha, \alpha)$ *innovations* (α *is a pre-chosen fixed "tuning" parameter). Here it is clear that if the chain is currently at* x*, a proposed new value* $x^\star$ *should be accepted with probability* $\min\{1, \phi(x^\star)/\phi(x)\}$*, where* $\phi(\cdot)$ *is the standard normal density.*

The full parallel program to implement the above example is given below.

```
#include <gsl/gsl_rng.h>
#include "gsl-sprng.h"
#include <gsl/gsl_randist.h>
#include <mpi.h>

int main(int argc,char *argv[])
{
  int k,i,iters; double x,can,a,alpha; gsl_rng *r;
  FILE *s; char filename[15];
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&k);
  if ((argc != 3)) {
    if (k == 0)
      fprintf(stderr,"Usage: %s <iters> <alpha>\n",argv[0]);
    MPI_Finalize(); return(EXIT_FAILURE);
  }
  iters=atoi(argv[1]); alpha=atof(argv[2]);
  r=gsl_rng_alloc(gsl_rng_sprng20);
  sprintf(filename,"chain-%03d.tab",k);
  s=fopen(filename,"w");
  if (s==NULL) {
    perror("Failed open");
    MPI_Finalize(); return(EXIT_FAILURE);
  }
  x = gsl_ran_flat(r,-20,20);
  fprintf(s,"Iter X\n");
  for (i=0;i<iters;i++) {
    can = x + gsl_ran_flat(r,-alpha,alpha);
```

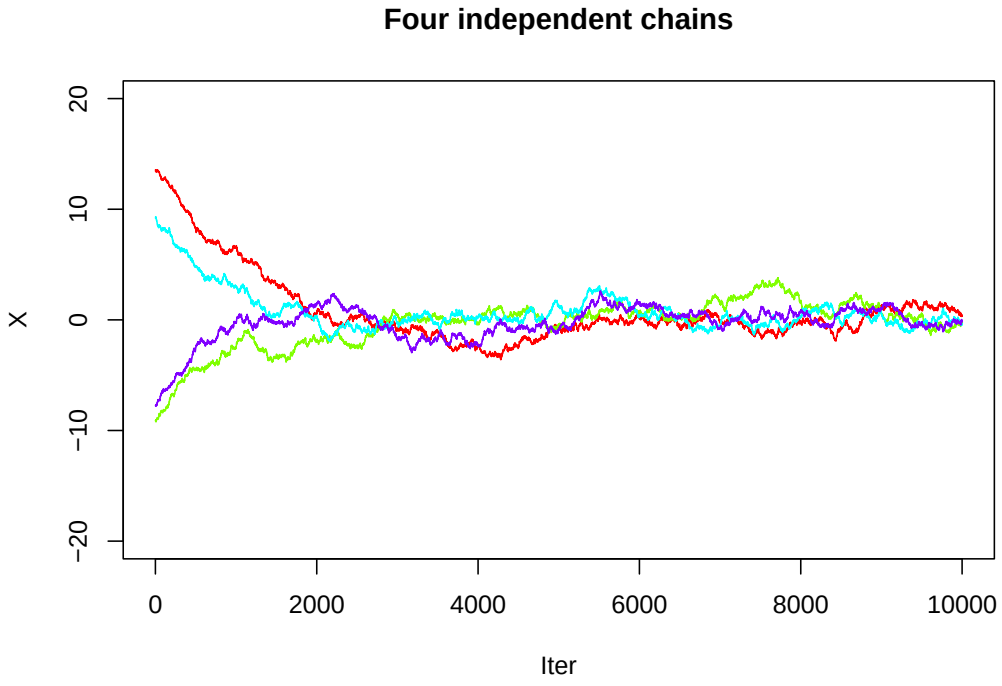

Figure 18.4: Trace plot of the output obtained by running the example parallel MCMC chains program for 10,000 iterations on each of four processors with an innovation parameter of $\alpha = 0.1$.

```
28      a = gsl_ran_ugaussian_pdf(can) / gsl_ran_ugaussian_pdf(x);
29      if (gsl_rng_uniform(r) < a)
30        x = can;
31      fprintf(s,"%d %f\n",i,x);
32    }
33    fclose(s);
34    MPI_Finalize(); return(EXIT_SUCCESS);
35  }
```

The first thing to note about the program is that lines 24 to 32 contain the essential part of the algorithm, which may be coded in exactly the same way in the serial context. Lines 1 to 23 set up the parallel processes, and create separate MCMC output files for each process. This program also illustrates how to trap errors and exit politely from an MPI program when an error is encountered. Figure 18.4 shows the result of running this program on four processors with parameters 10000 and 0.1.

An important consideration in the parallel context is that of the starting point of the chain. In principle, provided that a suitable burn-in period is used, the starting point does not matter, and hence any starting point could be used (indeed, all of the chains could be started off at the same point). On the other hand, if each chain could be started with an independent realisation from the target distribution, then no burn-in at all would be necessary, and parallelisation would be "perfect". Rosenthal (2000) suggests that where possible, an exact sampling technique such as coupling from the past (Propp and Wilson 1996) should be used on each processor in order to generate the first sample, and then

the chain run from these starting points, keeping all iterates. Unfortunately it is difficult to implement exact sampling for many complex models, and so a pragmatic solution is to initialise each chain at a random starting point that is in some sense "over-dispersed" relative to the target, and then burn-in until all chains have merged together. Note that the arguments sometimes put forward suggesting that "burn-in is pointless" only hold in the "one-long-run" context; in the context of parallel chains starting points are an issue (Rosenthal 2000).

In order to make things more concrete, consider a problem where preliminary runs have been used to ascertain that a burn-in of b iterations are required, followed by a main monitoring run of n iterations. In the serial context, a chain of length $b+n$ is required. It is usually reasonable to assume that each iteration takes roughly the same amount of processor time to compute, so that iterations may be used as a unit of time. If N processors are available to run parallel chains, then the iterations for the main monitoring run may be divided among them, so that $b + n/N$ iterations are required on each processor. This gives a speed-up of

$$\text{SpeedUp}(N) = \frac{b+n}{b+\frac{n}{N}} \quad \xrightarrow[\infty]{N} \quad \frac{b+n}{b},$$

meaning that the potential speed-up is limited for $b > 0$. Of course, when N processors are available, then the potential speed-up offered in principle for a "perfectly parallel" solution is N (attained here in the special case $b = 0$), so when b is a sizeable proportion of n, the actual speed-up attained falls well short of the potential. As the burn-in time is related to the mixing rate of the chain, and the mixing rate is related to the length of the main monitoring run required, b and n are often related. In many cases, $n = 10b$ is a useful rule-of-thumb. In this case we have

$$\text{SpeedUp}(N) = \frac{11}{1+\frac{10}{N}} \quad \xrightarrow[\infty]{N} \quad 11,$$

making a parallel chains approach quite attractive on small clusters. In particular, SpeedUp(8) ≈ 5. However, SpeedUp(16) < 7, and so effective utilisation of large clusters is not possible (Figure 18.5).

As with any MCMC-based approach to Bayesian computation, processing of the raw output is a vital part of the analysis. The most commonly used systems for output analysis are CODA, a library for the S-PLUS statistical language, and R-CODA, a version of CODA for R, the free S-PLUS alternative. Fortunately both CODA and R-CODA have excellent support for analysis of parallel chains, and hence can be used without any modification in this context.

Before moving on to the issues which arise when a parallel chains approach is felt not to be appropriate, it is worth considering the popular user-friendly MCMC software, WinBUGS (Lunn et al. 2000). This software allows a user to simply specify a model and prior using a graphical language, and then import data and run an MCMC algorithm for the problem without any programming. Consequently WinBUGS is extremely popular, particularly with applied statisticians. Early versions of WinBUGS offered no facilities for automated control or scripting, and hence were difficult to consider using in a parallel environment. However, as of version 1.4, WinBUGS offers limited remote scripting facilities which now make it realistic to consider using on a parallel cluster. Unfortunately this approach is not without problems. The most obvious problem is that WinBUGS runs only on Microsoft Windows platforms, whereas many dedicated parallel clusters are Unix-based; on the other hand, many universities have large student PC clusters running Microsoft Windows which stand

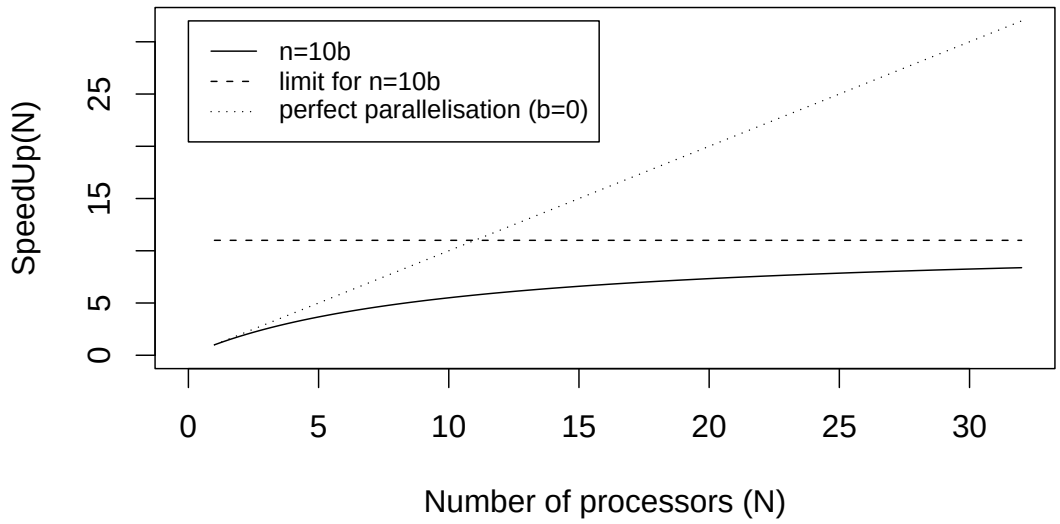

Figure 18.5: Graph showing the speedup curve for a parallel chains approach to parallelisation in the case $n = 10b$. The limiting speedup is also plotted, along with the curve (a straight line) for a perfect parallelisation (attained for $b = 0$).

idle for much of the time, and so a large number of Windows machines are potentially available for parallel computation if configured so to do. Another problem is that WinBUGS currently does not have a parallel random number generator, and hence parallel chains rely on the "different seeds" approach, which does not guarantee independence of the resulting chains. The main problem with using WinBUGS for parallel chains however (and it is a problem in the serial context too) is that WinBUGS currently uses a simple one-variable-at-a-time Gibbs or Metropolis-within-Gibbs sampler which leads to very poor mixing for many high dimensional problems. This in turn leads to very long burn-in times which makes a parallel chains approach to parallelisation very unattractive. However, the interested reader is referred to PyBUGS, a Python library which facilitates parallel WinBUGS sessions.

18.4.3 Blocking and marginalisation

It is tempting to try to separate the issue of parallel computation from that of MCMC algorithm design, but there are at least two reasons why this is misguided. The first is that typically one is interested in parallel strategies because the problem under consideration is of considerable importance and computational complexity. What is the point of parallelising a bad MCMC algorithm and running it on a powerful computing cluster if a little thought could give an algorithm which performs as well run on a standard desktop PC? Of course if the hardware is available, then one could still parallelise the new algorithm if further speed-up is still desirable. The second reason is that different algorithms parallelise in different ways. As has already been demonstrated, for a sophisticated MCMC algorithm with good mixing properties, a parallel chains approach has much to recommend it. However, many simple algorithms with poor mixing properties can often be speeded up by parallelising a

single chain (Section 18.4.4). Consequently the issues of algorithm design and parallelisation strategies must be considered together.

If a given Markov chain has poor mixing and long burn-in times, then a straightforward parallel chains approach is unlikely to be sensible. In this case the obvious solution is to work on parallelising the computations involved in running a single chain (Section 18.4.4), but another possibility is to develop a more sophisticated MCMC algorithm with better mixing properties. Usually individual iterations are computationally more expensive with better performing algorithms, and so it is important to try and judge the relative performance of the algorithm using a criteria such as "effective sample size" per unit time. CODA and R-CODA contain routines to calculate "effective sample size" from MCMC output. Care must be taken with such an approach however, as measures of effective sample size based on MCMC output reflect the apparent mixing-rate of the output, and not the true mixing rate of the underlying Markov chain. It is possible for a poor algorithm to apparently mix well, but in fact be only exploring a small part of the true target distribution. Consequently, given two algorithms, one of which is very simple, with many components, the other of which is more sophisticated, with fewer components, the more sophisticated algorithm will almost always be preferred, even if the the simpler algorithm appears to give slightly more effective independent samples per unit time. It is clear that such issues are already somewhat difficult to trade-off in the serial context, but things are even less clear in the parallel context, as here it is desirable to construct the algorithm with parallelisation in mind.

In this section a brief overview will be given of two techniques for improving the mixing of MCMC algorithms that are often applicable in the context of large complex models, and discuss the use of such techniques in a parallel context. It is helpful to use the notation of Section 18.2.1, considering MCMC algorithms having the posterior $\pi(\sigma, \theta|y)$ as their unique equilibrium distribution. Note that one of the advantages of the MCMC approach is that it renders trivial the marginalisation process; to look at the marginal distribution of any function of σ and θ, simply apply that function to each iterate and study the resulting samples. A simple MCMC algorithm will typically divide up σ and θ into the full collection of univariate components, and sample from each in turn from a kernel that preserves the equilibrium distribution (such as the "full-conditional" or a Metropolis update). However, the large number of components in the sampler typically results in a poorly mixing chain. Usually there are positive partial correlations between the components of θ, and in this case grouping together elements of θ into "blocks" that are updated simultaneously will invariably improve the mixing of the chain (Roberts and Sahu 1997). That is, if the components of the chain are $\sigma_1, \sigma_2, \ldots, \sigma_p, \theta_1, \theta_2, \ldots, \theta_q$, it will typically be beneficial to ensure that p and q are as small as possible whilst still allowing sampling from the full-conditional of each block directly. Even if direct sampling from the full-conditionals is not possible, the above is still valid (at least empirically) if good Metropolis-Hastings jumping rules for the blocks can be found (that is, rules which induce good mixing for the block in question, conditional on all other blocks). In the limiting case of $p = q = 1$, if exact sampling from $\pi(\sigma|\theta, y)$ and $\pi(\theta|\sigma, y)$ are possible, then the result is the classic "data augmentation" technique of Tanner and Wong (1987). For many problems, sampling from $\pi(\sigma|\theta, y)$ will be relatively straightforward, based on semi-conjugate updates, or simple Metropolis-Hastings kernels. Updating $\pi(\theta|\sigma, y)$ is typically more problematic. However, in a variety of situations (especially for models where $\theta, y|\sigma$ is linear Gaussian), local computation techniques (Section 18.2.3) can be used to accomplish this step (Rue 2001; Wilkinson and Yeung 2002; Wilkinson and Yeung 2003). In the linear Gaussian case, if the model is DAG based, the free software library `GDAGsim`

can be used to carry out the local computations, and if it is derived from a Markov random field, the library `GMRFlib` can be used instead.

Markov chains based on data augmentation can still be slow to mix if there is strong posterior dependence between σ and θ. In such cases, a carefully constructed Metropolis-Hastings method is sometimes the only practical way to implement a satisfactory MCMC scheme. See Tierney (1994) for an overview of the range of techniques available. One approach is to focus on the marginal posterior density $\pi(\sigma|y)$ (18.2). It is not usually possible to sample from this directly (otherwise MCMC would not be necessary), but a Metropolis-Hastings scheme can be constructed by proposing a new $\sigma^\star$ from a transition density $f(\sigma^\star|\sigma)$. The proposal is accepted with probability $\min\{1, A\}$, where

$$A = \frac{\pi(\sigma^\star)p(y|\sigma^\star)f(\sigma|\sigma^\star)}{\pi(\sigma)p(y|\sigma)f(\sigma^\star|\sigma)}.$$

This involves the marginal likelihood $p(y|\sigma)$, but this is usually tractable using local computation (Section 18.2.3) and the BMI (18.4) if sampling from $\pi(\theta|\sigma, y)$ is possible (Wilkinson and Yeung 2003). The technique of integrating out the latent process is sometimes referred to as *collapsing* the sampler, and is extremely powerful in a variety of contexts; see for example García-Cortés and Sorensen (1996) for an application in statistical genetics. Garside and Wilkinson (2003) consider application of this technique to lattice-Markov spatio-temporal models; the sampler which results has mixing properties that are sufficiently good to make a parallel chains approach to parallelisation extremely effective. Although other approaches to parallel MCMC for this model are possible (eg. parallelisation of a single chain for a simple Gibbs sampler), empirical evidence suggests that the combination of sophisticated block sampler and parallel chains results in the most effective use of the available hardware when many processors are available.

The marginal updating schemes are closely related to a class of single-block updating algorithms for $\pi(\sigma, \theta|y)$. Here the proposal is constructed in two stages. First $\sigma^\star$ is sampled from some transition density $f(\sigma^\star|\sigma, \theta)$, and then $\theta^\star$ is sampled from $\pi(\theta^\star|\sigma^\star, y)$. This proposed joint update is then accepted with probability $\min\{1, A\}$, where

$$A = \frac{\pi(\sigma^\star)p(y|\sigma^\star)f(\sigma|\sigma^\star, \theta^\star)}{\pi(\sigma)p(y|\sigma)f(\sigma^\star|\sigma, \theta)}.$$

Clearly this single-block updating strategy is equivalent to the marginal updating strategy if the proposal density for σ is independent of θ.

Of course in many situations direct sampling from $\pi(\theta|\sigma, y)$ will not be possible, but single-block samplers may still be constructed if this distribution may approximated by one for which sampling is possible. Consider again a proposal constructed in two stages. As before, a candidate $\sigma^\star$ is first sampled from some $f(\sigma^\star|\sigma, \theta)$. Next a candidate $\theta^\star$ is sampled from $g(\theta^\star|\sigma^\star, \theta)$. The proposed values are then accepted jointly with probability $\min\{1, A\}$, where

$$A = \frac{\pi(\sigma^\star)\pi(\theta^\star|\sigma^\star)p(y|\sigma^\star, \theta^\star)f(\sigma|\sigma^\star, \theta^\star)g(\theta|\sigma, \theta^\star)}{\pi(\sigma)\pi(\theta|\sigma)p(y|\sigma, \theta)f(\sigma^\star|\sigma, \theta)g(\theta^\star|\sigma^\star, \theta)}.$$

In the special case where $g(\theta^\star|\sigma^\star, \theta) = \pi(\theta^\star|\sigma^\star, y)$ this reduces to the single-block sampler previously discussed. The above form of the acceptance probability is the form most useful for implementation purposes. However, in order to gain insight into its form, it is helpful to re-write it as

$$A = \frac{\pi(\sigma^\star, y)}{\pi(\sigma, y)} \times \frac{f(\sigma|\sigma^\star, \theta^\star)}{f(\sigma^\star|\sigma, \theta)} \times \frac{\pi(\theta^\star|\sigma^\star, y)}{g(\theta^\star|\sigma^\star, \theta)} \times \frac{g(\theta|\sigma, \theta^\star)}{\pi(\theta|\sigma, y)}.$$

It is then clear that for relatively small proposed changes to σ, A depends mainly on how well $g(\theta^\star|\sigma^\star,\theta)$ approximates $\pi(\theta^\star|\sigma^\star,y)$. A good example of a situation where this technique is useful arises when $\pi(\theta^\star|\sigma^\star,y)$ is not tractable, but is well approximated by a linear Gaussian system which is (Knorr-Held and Rue 2002). See also Section 18.5 for a related technique.

18.4.4 Parallelising single chains

In the context of a parallel computing environment, the strategies outlined in the previous section may lead to a sampler that mixes sufficiently well that a parallel chains approach to parallelisation is quite adequate. Given that this is very simple to implement and scales well with the number of available processors (provided burn-in times are short), this is usually the strategy to strive for in the first instance. However, there are many situations (eg. intrinsically non-linear high dimensional latent process models), where the best practically constructible samplers mix poorly and have long burn-in times. In such cases it is desirable to take an algorithm for generating a single chain and run it on a parallel computer in order to speed up the rate at which iterations are produced. It is worth noting however, that even in this situation, it may be possible to improve the sampler by constructing an improved algorithm based on a non-centred parameterisation (NCP) of the problem, following the techniques of Papaspiliopoulos et al. (2003). It turns out that NCPs can be parallelised in the same way as other algorithms, so we will not give further details here.

Clearly Markov chain simulation is an iterative process; simulation of the next value of the chain cannot begin until the current value has been simulated*. However, for high-dimensional problems every iteration is a computationally intensive task, and there is often scope for parallelisation of the computation required for each.

If the MCMC algorithm is based on one of the sophisticated samplers from the previous section, which at each stage involve the updating of only one or two very large blocks, then parallelisation strategies must focus on speeding up the computations involved in those updates. Parallelisation of local computation algorithms has already been discussed in Section 18.2.4. For example, if θ is sampled using a local computation algorithm, then a parallel updating strategy must be adopted; see Pearl (1988) and Cowell et al. (1999) for further details, but again note that the speed-up provided by graph-propagation algorithms in the statistical context tends not to scale well as the number of available processors increases. If the system is linear Gaussian conditional on σ, then an approach based on large (probably sparse) matrices has probably been used for sampling and/or the computation of marginal likelihoods required in the acceptance probabilities. These computations can be speeded up by using parallel matrix algorithms. There is an issue of whether or not direct or iterative matrix algorithms must be used; generally speaking, direct algorithms must be used if samples from θ are required, but iterative algorithms will suffice if θ has been marginalised out of the problem completely. Note that parallel matrix algorithms are covered in detail elsewhere in this volume, to which the reader is referred for further details.

For high dimensional non-linear problems, a sampler based on many components is more likely to be effective. Once again it will be assumed that the components of the chain are $\sigma_1,\sigma_2,\ldots,\sigma_p,\theta_1,\theta_2,\ldots,\theta_q$, and that each block is updated once per iteration using a kernel

*There is actually an exception to this; where a pure Metropolis-Hastings *independence sampler* is being used, the next proposed value may be simulated and work may start on evaluation of the next acceptance probability before the current iteration is complete. In practice however, this is not much help, as pure independence samplers tend to perform poorly for large non-linear models.

which preserves the desired target distribution $\pi(\sigma, \theta|y)$. Typically the updates of σ will be very fast (given some sufficient statistics regarding the current state of θ), and the updates of θ will be very slow. Consequently, a serial MCMC algorithm will spend almost all of its time carrying out the updating of θ. If a parallel computing environment is available it is therefore natural to try and speed up the algorithm by parallelising the θ update step.

Parallelisation of the update of θ depends crucially on the conditional independence structure of the model (Section 18.2.2). The structure of interest is the undirected conditional independence graph for $\theta_1, \theta_2, \ldots, \theta_q$ conditional on both σ and y. Assume first the simplest possible case, where $\theta_i \perp\!\!\!\perp \theta_j|\sigma, y,\ i \neq j$. If this is the case then the update of any particular θ_i will not depend on the state of any other θ_j $(j \neq i)$. Then clearly each θ_i can be updated in parallel, so the q blocks of θ can be evenly distributed out to the N available processors to be updated in parallel. The essential details of each iteration of the MCMC algorithm can then be described as follows.

1. Each processor (k):
   (a) sequentially updates the q_k θs that have been assigned to it, using the current value of σ
   (b) computes summary statistics for the new θs that will be required to update σ
   (c) passes the summary statistics back to the root process (master program)
2. The root process combines the summary statistics in order to obtain an explicit form for the updating of σ
3. A new σ is sampled and output
4. The new σ is distributed out to each process

In Section 18.3.6 it has already been demonstrated how to gather and summarise statistics to a root processor, using `MPI_Reduce`. The only new feature required for the implementation of the above algorithm is the ability to "broadcast" a statistic from the root process to the slave processes. This is accomplished with the function `MPI_Bcast`, which has a similar syntax to `MPI_Reduce`. For example, the command

```
MPI_Bcast(sigma,p,MPI_DOUBLE,0,MPI_COMM_WORLD);
```

will take a `p`-vector of "doubles" called `sigma` defined on the root process 0 (but also defined and allocated on every other process), and distribute the current values of `sigma` on process 0 to every other process.

The above strategy will often work quite well when the θs are all independent of one another. Unfortunately this is rarely the case in practice. Consider next the case of serial dependence between the blocks, as depicted in Figure 18.6. Here we have the property $\theta_i \perp\!\!\!\perp \theta_j|\theta_{i-1}, \theta_{i+1},\ i \neq j$. The update of θ_i therefore depends on its neighbours θ_{i-1} and θ_{i+1} meaning that the θ can not all be updated in parallel. However, all of the odd-numbered blocks are independent of one another given the even-numbered blocks (together with σ and y). So, breaking the updating of θ into two stages, all of the odd-numbered blocks may be updated in parallel conditional on the even numbered blocks. When this is complete, the even-numbered blocks may be updated in parallel conditional on the odd-numbered blocks. See Section 18.5 for an example of this approach in practice.

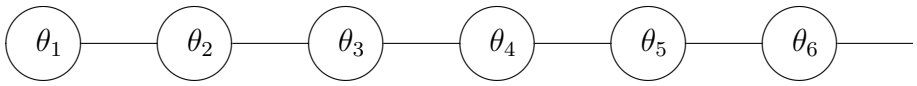

Figure 18.6: Undirected conditional independence graph for the case of serial dependence between the θs (conditional on σ and y)

The same basic strategy can be applied when the conditional independence structure is more complex. Using the terminology of Winkler (1995) (to which the reader is referred for more details), let $T = \{T_1, T_2, \ldots, T_c\}$ be a partition of $\{\theta_1, \theta_2, \ldots, \theta_q\}$, where each T_i is a set of blocks that are mutually independent given the remaining blocks (eg. the odd-numbered blocks from the previous example). The smallest value of c for which it is possible to construct such a partition is known as the *chromatic number* of the graph (eg. for the example of serial dependence, the chromatic number of the graph is 2). Incidentally, this is the minimum number of colours required to paint the nodes of the graph so that no two adjacent nodes are the same colour. It is clear that the blocks allocated to each T_i can be updated in parallel, and so it is clearly desirable to make each T_i as large as possible, and c as small as possible. It should be noted that for completely general conditional independence graphs, finding good partitions is an NP-hard combinatorial optimisation problem; however, for most statistical applications, it tends to be quite straightforward to construct good partitions "by hand". Consider, for example, a four-neighbour Markov random field model based on a 2d regular square lattice (Figure 18.7). Here again the chromatic number is just 2; if the graph is painted as a "chequer-board" the black nodes are mutually independent given the white nodes and vice versa.

To implement the parallel updating in practice, the blocks in each T_i are distributed evenly between the available processors so that they can be updated in parallel. In general then, the computations carried out at each iteration of the parallel MCMC algorithm will be as follows.

1. Each processor (k):
   (a) For each i in 1 to c:
      i. sequentially updates all the blocks in T_i allocated to it
      ii. distributes necessary state information regarding updates to adjacent processors
      iii. receives such information from adjacent processors
   (b) computes summary statistics for the new θs that will be required to update σ
   (c) passes the summary statistics back to the root process (master program)
2. the root process combines the summary statistics in order to obtain an explicit form for the updating of σ
3. a new σ is sampled and output
4. the new σ is distributed out to each process

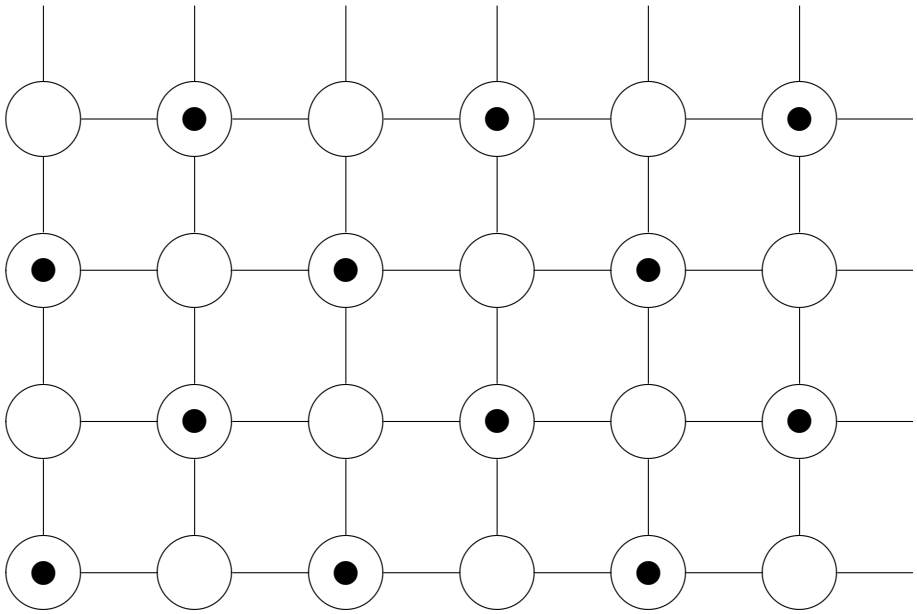

Figure 18.7: Undirected conditional independence graph for a four-neighbour Markov random field on a regular square lattice

The only new features required to implement this algorithm concern the exchange of information between processors. Here, the functions `MPI_Send` and `MPI_Recv` can be used to send on one processor and receive on another. The function `MPI_Sendrecv` is also useful for exchanging information between a pair of processors. See Snir et al. (1998) for precise syntax and further details.

In fact the above algorithms have made a simplifying assumption regarding the update of the parameter set σ. The above algorithms work as described if σ is univariate, or if it is convenient to directly update all of σ as a single block. Note that if the elements of $\sigma|\theta, y$ are conditionally independent of one another, then updating each in turn and updating in a single block are equivalent, so this case occurs quite often in practice. However, it is also the case that often the p components of σ are not conditionally independent, and are most conveniently updated separately. Here the above algorithm is not adequate, as the summaries required for the update of σ_j may depend on the result of the update for σ_i $(i < j)$. In this situation a revised algorithm is required.

1. Each processor (k):
   (a) for each i in 1 to c:
      i. sequentially updates all the blocks in T_i allocated to it
      ii. distributes necessary state information regarding updates to adjacent processors
      iii. receives such information from adjacent processors
2. for each i in 1 to p:
   (a) each processor (k):

i. computes summary statistics that will be required to update σ_i

ii. passes the summary statistics back to the root process

(b) the root process combines the summary statistics in order to obtain an explicit form for the updating of σ_i

(c) a new σ_i is sampled and output

(d) the new σ_i is distributed out to each processor

This algorithm involves a greater number of passed messages than the previous algorithm, so the latency associated with each passed message will make this algorithm less efficient than the previous when both are valid. No additional MPI commands are required to implement this scheme.

Note that a fundamental limitation of these so-called "partially parallel" algorithms is that they cannot provide an N-fold speed-up on N processors even for the θ-updating step due to the synchronisation requirements. There are various "check points" where further progress can not be made by one processor if every other processor has not reached the same point. The very tight synchronisation requirements make careful load-balancing key. Even then, because not all processors are not carrying out *exactly* the same computations (even if the are executing essentially the same sequence of function calls), processors stand idle until the last one "catches up". This has serious implications for the extent to which such algorithms will scale up to large numbers of processors. Also note that communication overheads can become an issue if processors need to exchange a lot of information regarding the current state of θ. Obviously it is desirable to construct the partition T and the allocation of blocks to processors in such a way as to minimise the inter-processor communication. See Whiley and Wilson (2002) for a thorough investigation of these issues in the context of spatial latent Gaussian models with Poisson count data.

18.5 Case study: Stochastic volatility modelling

18.5.1 Background

This section is concerned with parallelisation strategies for Bayesian inference in a discrete-time univariate stochastic volatility model. This problem is high-dimensional and intrinsically non-linear, so even "good" MCMC algorithms tend to exhibit poor mixing and long burn-in times, making a parallel chains approach apparently unattractive. Therefore parallelisation of a single chain will be explored, and its performance relative to the serial and parallel chains implementations examined as a function of the number of available processors.

The log-Gaussian stochastic volatility model is considered, as examined in Jaquier et al. (1994), though here it is presented using the notation of Pitt and Shephard (1999). The block-updating algorithm used is similar to that of Shephard and Pitt (1997) which uses a simulation smoother (de Jong and Shephard 1995) in order to construct proposed updates for blocks which are then accepted/rejected using a Metropolis-Hastings step. It is worth noting that there is another strategy for constructing an efficient sampler for this model, as described in Kim et al. (1998), but such an approach will not be considered here.

The basic model describes a Gaussian noise process y, which has zero mean, but has variance (volatility) varying through time according to a mean-reverting stationary stochastic

process. More precisely

$$y_t = \varepsilon_t \exp\{\alpha_t/2\}, \ t = 1, 2, \ldots, n \tag{18.7}$$
$$\alpha_t = \mu + \phi(\alpha_{t-1} - \mu) + \eta_t, \tag{18.8}$$

where ε and η are mean zero Gaussian white noise processes with variances 1 and σ_η^2 respectively. This model is written $y \sim ISV_n(\phi; \sigma_\eta; \mu)$. It is common practice to initialise the (log) volatility process α with the stationary distribution: $\alpha_0 \sim N(\mu, \sigma_\eta^2/\{1-\phi^2\})$. In terms of the notation from Section 18.2, we have $\sigma = (\phi, \sigma_\eta, \mu)$ and $\theta = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ with data y as before. Interest is in the joint posterior $\pi(\sigma, \theta|y)$, but for this to be completely specified, $\pi(\sigma)$ is also required. For illustrative purposes, an independent semi-conjugate specification is adopted:

$$\mu \sim N(\mu_\mu, \sigma_\mu^2), \quad \phi \sim N(\mu_\phi, \sigma_\phi^2), \quad \sigma_\eta^{-2} \sim \Gamma(\alpha_\sigma, \beta_\sigma).$$

Note however that this is not necessarily optimal. In particular, a Beta prior for ϕ is very often used in practice to describe an appropriate informative prior on this parameter.

18.5.2 Basic MCMC scheme

A simple one-variable-at-a-time sampler for this model has $3 + n$ components, and exhibits very poor mixing for large n. It is therefore desirable to block together sequences of successive α_ts in order to reduce the number of components in the sampler to $3+q$ ($q << n$), where then $\theta = (\theta_1, \theta_2, \ldots, \theta_q)$, and hence improve mixing. This is only likely to be beneficial, however, if a good updating mechanism can be found for the update of each of the components θ_i. This is achieved by approximating the joint distribution of θ and y as linear Gaussian, and then using linear Gaussian local computation techniques to sample an approximation to the full conditional for each θ_i, which is then corrected using a Metropolis-Hastings step. The linear Gaussian approximation is constructed by re-writing (18.7) as

$$\log(y_t^2) = \alpha_t + \log(\varepsilon_t^2). \tag{18.9}$$

The random quantity $\log(\varepsilon_t^2)$ is not Gaussian, but its mean and variance may be computed as -1.27 and $\pi^2/2$, respectively. Now if $\log(\varepsilon_t^2)$ is approximated by a $N(-1.27, \pi^2/2)$ random quantity, then (18.9) and (18.8) together form a linear Gaussian system in the form of a dynamic linear model (West and Harrison 1997). Local computation algorithms applied to dynamic linear models are generally referred to as Kalman filters. Here the simulation smoother (de Jong and Shephard 1995) will be used to sample from each block θ_i based on the linear Gaussian approximation. The log-likelihood for a sampled block under this approximate model is

$$l_G(\theta_i) = -\frac{3n}{2}\log\pi - \frac{1}{\pi^2}\sum(\log y_t^2 + 1.27 - \alpha_t)^2,$$

where the sum is over those α_t in block θ_i. Under the true model, however, the log-likelihood is

$$l(\theta_i) = -\frac{n}{2}\log(2\pi) - \frac{1}{2}\sum(\alpha_t + y_t^2 \exp\{\alpha_t\}).$$

Therefore the proposed new block $\theta_i^\star$ should be accepted as a replacement of the current θ_i with probability $\min\{1, A\}$, where

$$\log A = [l(\theta_i^\star) - l_G(\theta_i^\star)] - [l(\theta_i) - l_G(\theta_i)].$$

This step corrects the linear Gaussian approximation, keeping the exact posterior distribution as the equilibrium of the Markov chain. Fixed block sizes of m will be assumed, and it will be assumed that $n = 2rmN$ where N is the number of processors and r is an integer. In the serial context $N = 1$ is assumed.

Updating the elements of σ is particularly straightforward due to the semi-conjugate prior specification. Note that whilst for many models, the elements of $\sigma|\theta, y$ are conditionally independent, this is not the case here. One consequence of this is that there is potential for improving mixing of the chain by using a block update for the elements of σ (though this is not considered here). Another consequence is that necessary sufficient statistics required for the updating of the elements of σ must be computed as they are required. They cannot all be computed and passed back to the root process in one go after the update of the latent process (which would be desirable, due to the latency associated with inter-process communication). Standard Bayesian calculations lead to the following full-conditionals for the elements of σ:

$$\mu|\cdot \sim N\left(\frac{\dfrac{\mu_\mu}{\sigma_\mu^2} + \dfrac{1-\phi}{\sigma_\eta^2}\displaystyle\sum_{i=2}^{n}(\alpha_i - \phi\alpha_{i-1})}{\dfrac{1}{\sigma_\mu^2} + \dfrac{(n-1)(1-\phi)^2}{\sigma_\eta^2}}, \frac{1}{\dfrac{1}{\sigma_\mu^2} + \dfrac{(n-1)(1-\phi)^2}{\sigma_\eta^2}}\right) \tag{18.10}$$

$$\phi|\cdot \sim N\left(\frac{\dfrac{\mu_\phi}{\sigma_\phi^2} + \dfrac{1}{\sigma_\eta^2}\displaystyle\sum_{i=2}^{n}(\alpha_{i-1} - \mu)(\alpha_i - \mu)}{\dfrac{1}{\sigma_\phi^2} + \dfrac{1}{\sigma_\eta^2}\displaystyle\sum_{i=2}^{n}(\alpha_i - \mu)^2}, \frac{1}{\dfrac{1}{\sigma_\phi^2} + \dfrac{1}{\sigma_\eta^2}\displaystyle\sum_{i=2}^{n}(\alpha_i - \mu)^2}\right) \tag{18.11}$$

$$\sigma_\eta^{-2} \sim \Gamma\left(\alpha_\sigma + \frac{1}{2}(n-1), \beta_\sigma + \frac{1}{2}\sum_{i=2}^{n}\left\{(\alpha_i - \mu) - \phi(\alpha_{i-1} - \mu)\right\}^2\right). \tag{18.12}$$

In each case the (very small) contribution to the likelihood due to α_1 has been neglected.

18.5.3 Parallel algorithm

As always, adaptation of the serial algorithm to parallel chains is trivial. Construction of a parallel algorithm for a single chain requires a little more work. However, there is a simple serial dependence for the blocks of θ, as shown in Figure 18.6. We therefore use the assumed relationship $n = 2rmN$ giving $2rN$ blocks for θ. The first $2r$ blocks are assigned to the first processor, the next $2r$ to the second processor, *etc.*, so that r "odd" and r "even" blocks are assigned to each processor. The basic algorithm is then as described in Section 18.4.4.

As well as storing the blocks allocated to it, each processor also needs to know something of the blocks adjacent to the blocks allocated to it. In this case, all that is required is the very last α value assigned to the previous processor, and the very first α value assigned to the subsequent processor, keeping communication overheads fairly minimal. Assuming that the processes are all initialised in a self-consistent way, each processor passes its first α value down to the previous processor after updating the "odd" blocks, and passes its last α value up to the subsequent processor after updating the "even" blocks. The situation is illustrated in Figure 18.8.

It is clear from the form of the full-conditionals for the model parameters (18.10)–(18.12)

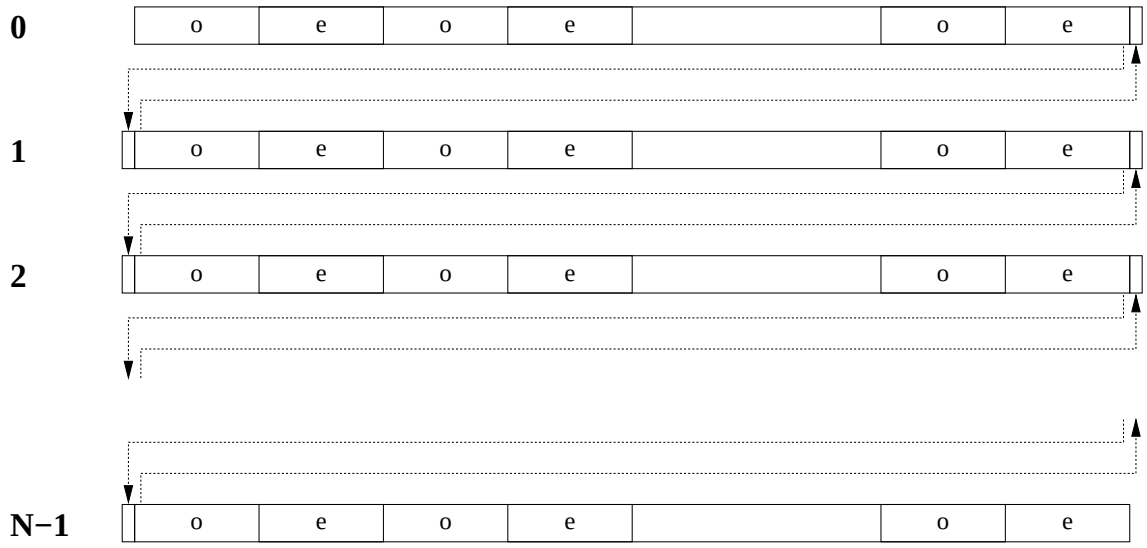

Figure 18.8: Plan of the allocation of the θ blocks to the available processes, and the messages passed between them

that the summary statistics required for parameter updating are

$$\begin{aligned}
\text{Summ}_1 &= \sum_{i=2}^{n} (\alpha_i - \phi\alpha_{i-1}) \\
\text{Summ}_2 &= \sum_{i=2}^{n} (\alpha_i - \mu)^2 \\
\text{Summ}_3 &= \sum_{i=2}^{n} (\alpha_{i-1} - \mu)(\alpha_i - \mu) \\
\text{Summ}_4 &= \sum_{i=2}^{n} \big(\{\alpha_i - \mu\} - \phi\{\alpha_{i-1} - \mu\}\big)^2.
\end{aligned}$$

Each processor therefore calculates these four summaries for the part of the θ process allocated to it, and passes them back to the root process where they are combined and then used to sample new parameters. The new parameters are then broadcast back to all processors, and the update of the θ process starts again. Note again, however, that the summaries are not all computed at the same time. Summ_1 is computed for the update of μ, then Summ_2 and Summ_3 are computed for the update of ϕ (using the new value of μ), then Summ_4 is computed for the update of σ_η (using the new values of μ and ϕ). This additional message-passing overhead is undesirable, but a necessary consequence of the lack of conditional independence of the elements of σ.

18.5.4 Results

In order to compare the performance of the serial and parallel algorithms, some data simulated from the true model will be used. A total of 3,200 observations were simulated using parameters $\mu = 1,\ \phi = 0.8,\ \sigma_\eta = 0.1$. The hyper-parameters used to complete the prior specification were

$$\mu_\mu = 0,\ \sigma_\mu = 100,\ \mu_\phi = 0.8,\ \sigma_\phi = 0.1,\ \alpha_\sigma = 0.001,\ \beta_\sigma = 0.001.$$

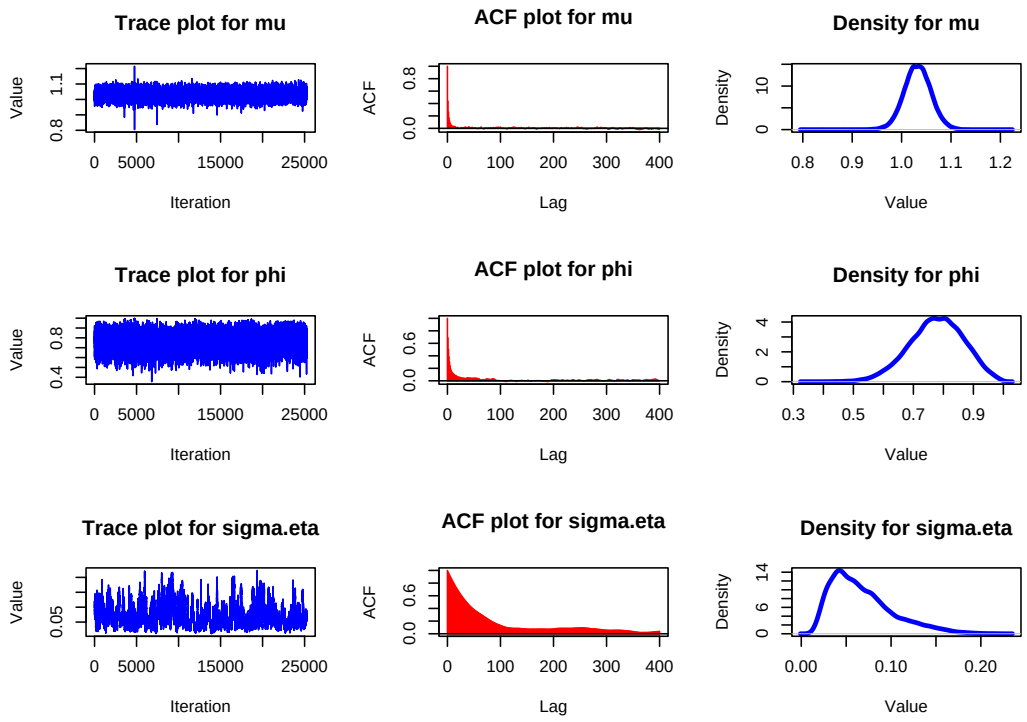

Figure 18.9: Analysis of output from the serial algorithm (thinned by a factor of 40), showing trace plots, auto-correlation plots and marginal density estimates for the three key parameters

This follows common practice for stochastic volatility modelling, where a fairly informative prior is used for ϕ, but weak priors are adopted for μ and σ_η. A block size of $m = 50$ was used for the MCMC algorithm. Initial values must be chosen for σ and θ. Although these are essentially arbitrary in principle, they actually have important implications for burn-in times, and hence are quite important in practice. Setting σ can be done either by setting the components to plausible values or by sampling from a range of plausible values. Once this has been done, θ can be set by sampling from the linear Gaussian approximation to $\theta|\sigma, y$. This ensures that θ is consistent with σ, and hence will not immediately "drag" σ away from the range of plausible values. In cases where it is less straightforward to initialise θ to be consistent with σ, burn-in times can be very much longer.

Preliminary runs suggest that a burn-in of $b = 10,000$ iterations should be used, followed by a monitoring run of $n = 1,000,000$ iterations. The results for running the serial algorithm are illustrated in Figure 18.9. In general the model does a good job of uncovering the true parameters; for example, they all are contained within 95% equitailed posterior probability intervals. It is clear however, that σ_η is slightly under-estimated and μ is a little over-estimated, which is a common experience using such models. There are various alternative prior specifications and model parameterisations that can be used to reduce this effect, but such issues are somewhat tangential to the current discussion, and will not be considered further here.

Despite the relatively long burn-in time, the fact that a such a long monitoring run is required means that the burn-in time is a fairly small proportion of the total running time. This means that despite initial appearances to the contrary, a parallel chains approach to

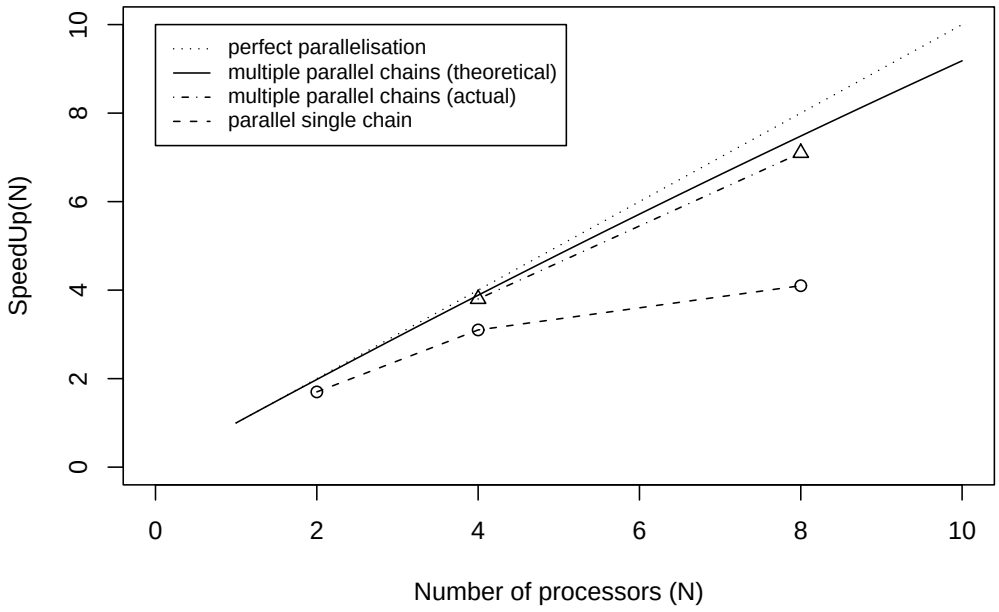

Figure 18.10: Graph showing how the speed-up varies with the number of processors for the two competing parallel algorithms in this particular example

this problem is well worth considering. Here the limiting speed-up is 101, which is very good (though a very large number of processors would be required to get close to this). For $N = 8$ processors, the speed-up is roughly 7.5, which represents close to perfect utilisation of the available hardware.

Running the parallel version of the algorithm for $1,010,000$ iterations resulted in speed-ups 1.7, 3.1 and 4.1 on 2, 4 and 8 processors, respectively. This is plotted together with the (theoretical and actual) parallel chains performance in Figure 18.10. The actual speed-ups obtained by executing a parallel-chains program on 4 and 8 processors were 3.8 and 7.1 respectively; close to (but less than) the theoretical values of 3.9 and 7.5. It is clear therefore, that in this example, even though the mixing is very poor, by initialising the chain carefully in order to minimise burn-in times, the parallel chains approach appears to be superior to an approach based on parallelising a single chain (and is much easier to program!). All of the ANSI `'C'` source code for the analysis of this example is available (together with the other simple example programs) from the author's web site.

18.6 Summary and Conclusions

A number of techniques for utilising parallel computing hardware in the context of Bayesian inference have been examined. For conventional (non MCMC) methodology, techniques based on parallelisation of local computation algorithms are most promising. In particular, parallel algorithms for large matrices are especially valuable in the context of large linear models (such techniques are explored in detail elsewhere in this volume). If a pure Monte Carlo solution to the problem can be found, then parallelisation of computations is very

straightforward provided a good parallel random number generator library (such as SPRNG) is used. The same applies to MCMC algorithms if a parallel chains approach is felt to be appropriate.

Poorly mixing MCMC algorithms with long burn-in times are not ideally suited to a parallel chains approach, due to the fact that the burn-in must be repeated on every available processor. However, as the case study illustrated, even where this is the case it often turns out that the length of monitoring run required is such that the burn-in time is a relatively small proportion making a parallel chains approach attractive. In the case study the burn-in time was small relative to the length of monitoring run due to the fact that it was relatively easy to initialise the latent process (θ) in a sensible way (using a sample from the linear Gaussian approximation). In cases where initialising the latent process well is less straightforward, burn-in times may not be such a small proportion of the required monitoring run. In such situations it is desirable to re-design the sampler if possible. If it is not possible so to do, it may be desirable to consider strategies for parallelising a single chain. If the algorithm contains large blocks of computation (such a calculations involving large matrices), then it may be possible to parallelise the sampler by using parallel libraries (such as a parallel matrix library). Alternatively, if the sampler is constructed by cycling through a large number of blocks of variables, the sampler may be parallelised by sampling carefully selected sets of blocks in parallel. Unfortunately this requires considerable modification of the serial algorithm, and does not always scale well as the number of processors increases, due to the very tight synchronisation requirements of MCMC algorithms.

MCMC algorithms have revolutionised Bayesian inference (and statistical inference for complex models more generally), rendering tractable a huge range of inference problems previously considered to be of impossible complexity. Of course statisticians are becoming ever more ambitious in the range of models they consider. Whilst MCMC algorithms for large complex stochastic models are now relatively straightforward to implement in principle, many require enormous amounts of computing power in order to execute. Consequently, effective exploitation of parallel hardware is of clear relevance to many Bayesian statisticians.

Acknowledgements

This research is supported by a grant from the UK Royal Society (Grant ref. 22873).

Bibliography

Brooks, S. P. (1998). Markov chain Monte Carlo method and its application. *The Statistician 47*(1), 69–100.

Chib, S. (1995). Marginal likelihood from the Gibbs output. *Journal of the American Statistical Association 90*(432), 1313–1321.

Cowell, R. G., A. P. Dawid, S. L. Lauritzen, and D. J. Speigelhalter (1999). *Probabilistic Networks and Expert Systems*. New York: Springer.

de Jong, P. and N. Shephard (1995). The simulation smoother for time-series models. *Biometrika 82*, 339–350.

Devroye, L. (1986). *Non-uniform Random Variate Generation*. New York: Springer Verlag.

Gamerman, D. (1997). *Markov Chain Monte Carlo*. Texts in Statistical Science. Chapman and Hall.

García-Cortés, L. A. and D. Sorensen (1996). On a multivariate implementation of the Gibbs sampler. *Genetics Selection Evolution 28*, 121–126.

Garside, L. M. and D. J. Wilkinson (2003). Dynamic lattice-Markov spatio-temporal models for environmental data. In J. M. Bernardo et al (Ed.), *Bayesian Statistics 7*, Tenerife, pp. 535–542. OUP.

Gelman, A. and D. Rubin (1992). Inference from iterative simulation using multiple sequences. *Statistical Science 7*, 457–511.

George, A. and J. W. Liu (1981). *Computer Solution of Large Sparse Positive Definite Systems*. Series in computational mathematics. Prentice-Hall.

Geyer, C. J. (1992). Practical Markov chain Monte Carlo. *Statistical Science 7*, 473–511.

Jaquier, E., N. G. Polson, and P. E. Rossi (1994). Bayesian analysis of stochastic volatility models. *Journal of Business and Economic Statistics 12*, 371–417.

Kim, S., N. Shephard, and S. Chib (1998). Stochastic volatility: likelihood inference and comparison with ARCH models. *Reviews Economic Studies 65*, 361–393.

Knorr-Held, L. and H. Rue (2002). On block updating in Markov random field models for disease mapping. *Scandanavian Journal of Statistics*. To appear.

Kontoghiorghes, E. J. (2000). *Parallel Algorithms for Linear Models: Numerical methods and estimation problems*, Volume 15 of *Advances in Computational Economics*. Boston: Kluwer.

Lauritzen, S. L. (1992). Propagation of probabilities, means, and variances in mixed graphical association models. *Journal of the American Statistical Association 87*(420), 1098–1108.

Lauritzen, S. L. (1996). *Graphical Models*. Oxford: Oxford Science Publications.

Lunn, D. J., A. Thomas, N. Best, and D. J. Spiegelhalter (2000). WinBUGS – a Bayesian modelling framework: Concepts, structure, and extensibility. *Statistics and Computing 10*(4), 325–337.

Mascagni, M. and A. Srinivasan (2000). SPRNG: A scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software 23*(3), 436–461.

O'Hagan, A. (1994). *Bayesian Inference*, Volume 2B of *Kendall's advanced theory of statistics*. London: Arnold.

Pacheco, P. S. (1997). *Parallel Programming with MPI*. San Francisco: Morgan Kaufmann.

Papaspiliopoulos, O., G. O. Roberts, and M. Sköld (2003). Non-centered parameterisations for hierarchical models and data augmentation (with discussion). In J. M. Bernardo et al (Ed.), *Bayesian Statistics 7*, Tenerife, pp. 307–326. OUP.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann.

Pitt, M. and N. Shephard (1999). Time varying covariances: a factor stochastic volatility approach. In J.-M. Bernardo et al (Ed.), *Bayesian Statistics 6*, Oxford, pp. 3–26. OUP.

Propp, J. G. and D. B. Wilson (1996). Exact sampling with coupled Markov chains and applications to statistical mechanics. *Random Structures and Algorithms 9*, 223–252.

Ripley, B. D. (1987). *Stochastic Simulation*. New York: Wiley.

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

Rosenthal, J. S. (2000). Parallel computing and Monte Carlo algorithms. *Far East Journal of Theoretical Statistics 4*, 207–236.

Rue, H. (2001). Fast sampling of Gaussian Markov random fields. *Journal of the Royal Statistical Society B:63*(2), 325–338.

Shephard, N. and M. K. Pitt (1997). Likelihood analysis of non-Gaussian measurement time series. *Biometrika 84*(3), 653–667.

Snir, M., S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra (1998). *MPI - The Complete Reference: Volume 1, The MPI Core* (Second ed.). Cambridge, Massachusetts: MIT Press.

Srinivasan, A., D. M. Ceperley, and M. Mascagni (1999). Random number generators for parallel applications. *Advances in Chemical Physics 105*, 13–36. Monte Carlo methods in chemical physics.

Tanner, M. A. and W. H. Wong (1987). The calculation of posterior distributions by data augmentation. *Journal of the American Statistical Association 82*(398), 528–540.

Tierney, L. (1994). Markov chains for exploring posterior distributions (with discussion). *Annals of Statistics 21*, 1701–1762.

West, M. and J. Harrison (1997). *Bayesian Forecasting and Dynamic Models* (second ed.). New York: Springer.

Whiley, M. and S. P. Wilson (2002). Parallel algorithms for Markov chain Monte Carlo methods in latent Gaussian models. Technical report, Department of Statistics, Trinity College Dublin. Submitted for publication.

Whittaker, J. (1990). *Graphical Models in Applied Multivariate Statistics*. Chichester: Wiley.

Wilkinson, D. J. and S. K. H. Yeung (2002). Conditional simulation from highly structured Gaussian systems, with application to blocking-MCMC for the Bayesian analysis of very large linear models. *Statistics and Computing 12*, 287–300.

Wilkinson, D. J. and S. K. H. Yeung (2003). A sparse matrix approach to Bayesian computation in large linear models. *Computational Statistics and Data Analysis XX*, xx–xx. In press.

Winkler, G. (1995). *Image Analysis, Random Fields and Dynamic Monte Carlo Methods*. Berlin Heidelberg: Springer.

Web references

Name	URL	Comment
Beowulf	http://www.beowulf.org/	Cluster computing
GDAGsim	http://www.staff.ncl.ac.uk/d.j.wilkinson/software/gdagsim/	
GMRFlib	http://www.math.ntnu.no/~hrue/GMRFLib/	Spatial library
GSL	http://sources.redhat.com/gsl/	GNU Scientific library
GSL-SPRNG	http://www.staff.ncl.ac.uk/d.j.wilkinson/software/	
LA Software	http://www.netlib.org/utk/people/JackDongarra/la-sw.html	
LAM	http://www.lam-mpi.org/	MPI implementation
LDP	http://www.tldp.org/	Linux documentation
MPI	http://www-unix.mcs.anl.gov/mpi/	Message-passing interface
MPICH	http://www-unix.mcs.anl.gov/mpi/mpich/	MPI implementation
PSPASES	http://www-users.cs.umn.edu/~mjoshi/pspases/	Parallel direct solver
PVM	http://www.epm.ornl.gov/pvm/	Parallel virtual machines
PyBugs	http://www.simonfrost.com/	Parallel WinBUGS
Python	http://www.python.org/	Scripting language
R Project	http://www.r-project.org/	Statistical language
R-CODA	http://www-fis.iarc.fr/coda/	MCMC Output analysis
ScaLAPACK	http://www.netlib.org/scalapack/	Parallel solver
SPRNG	http://sprng.cs.fsu.edu/	PPRNG library
WinBUGS	http://www.mrc-bsu.cam.ac.uk/bugs/	MCMC software

Contents