Parallel Markov Chain Monte Carlo Simulation by Pre-Fetching

A. E. BROCKWELL

In recent years, parallel processing has become widely available to researchers. It can be applied in an obvious way in the context of Monte Carlo simulation, but techniques for "parallelizing" Markov chain Monte Carlo (MCMC) algorithms are not so obvious, apart from the natural approach of generating multiple chains in parallel. Although generation of parallel chains is generally the easiest approach, in cases where burn-in is a serious problem, it is often desirable to use parallelization to speed up generation of a single chain. This article briefly discusses some existing methods for parallelization of MCMC algorithms, and proposes a new "pre-fetching" algorithm to parallelize generation of a single chain.

Key Words: Bayesian; Inference; Parallel processing.

1. INTRODUCTION

Over the last two decades, the increased availability of cheap computing power has dramatically altered the way statistical analyses are carried out. Many problems previously considered intractable can now be solved by intensive numerical methods. In addition, in recent years, networking has also become cheap. The vast majority of computers sold now come with built-in ethernet adaptors. Partly as a result of this, parallel computing has received new impetus, because it is possible to put together a group of networked computers for a cost on the order of $1,000 per machine. Such networks, typically consisting of machines without keyboards or displays, are now widespread, and commonly referred to as "Beowulf clusters." Furthermore, well-defined standards for communication between processors have been developed. The "message passing interface" (MPI, see, e.g., Gropp, Lusk, and Skjellum 1999) is now widely accepted, and implementations in the form of C/C++/Fortran libraries are publicly available. Thus, parallel processing can be exploited by dividing a task into a number of subtasks that can be executed in parallel. Each subtask is executed on a separate processor, and then the results are combined, typically by a main

A. E. Brockwell is Assistant Professor, Department of Statistics, Carnegie Mellon University, Pittsburgh, PA 15213-3890 (E-mail: abrock@stat.cmu.edu).

Journal of Computational and Graphical Statistics, Volume 15, Number 1, Pages 246–261
DOI: 10.1198/106186006X100579

"controlling" processor. Such a scheme can be implemented by writing code in C, C++, or Fortran, making use of the MPI library to handle interprocess communication. Alternatively, for a simpler higher level solution, one can use the recently developed "Snow" package for R (see, e.g., http://cran.r-project.org and the links therein to associated packages).

In statistics, perhaps the most obvious application of parallel processing is in Monte Carlo simulation, where one estimates

$$h_\pi = \int h(x) d\pi(x), \tag{1.1}$$

for some function h and probability distribution π. The Monte Carlo approximation is simply

$$\hat{h} = \frac{1}{t} \sum_{j=1}^{t} h(X_j), \tag{1.2}$$

where $\{X_j\}$ is a sequence of independent draws from the distribution π. Such a problem is trivial to parallelize. The basic principle is to subdivide the sum into $P \geq 2$ components, and assign one processor to evaluate each component. The results for each component can then be added and normalized, either by the user, or by the controlling processor, to obtain the final result. When interprocessor communication time is negligible compared to the time taken to execute each subtask, and processors are roughly the same speed (i.e., they are "balanced"), such an approach leads to an increase in speed by a factor approximately equal to P.

Markov chain Monte Carlo (MCMC) methods (see Gilks, Richardson, and Spiegelhalter 1996) are a variant of Monte Carlo schemes in which a Markov chain $\{X_j,\ j = 1, 2, \ldots\}$ with limiting distribution π is generated. It can be used for estimating posterior distributions in a wide class of models, even when the density of the distribution includes an unknown normalizing constant. Estimation can still be carried out using (1.2), but in this case, elements of the sequence $\{X_j\}$ are not independent of each other. Furthermore, the initial value is typically not a draw from the distribution π. However, if the chain is constructed properly, then $X_t \xrightarrow{d} \pi$, and under certain conditions, the estimator $\hat{h}$ converges to h_π as $t \to \infty$. Unfortunately, generation of a Markov chain is not well suited to be carried out by parallel processing. The process is fundamentally sequential in nature; the distribution of X_{j+1} depends on the value of X_j, so simulation for one step is not seemingly possible until the results for the previous step have been obtained. On the other hand, it is often highly desirable to speed up MCMC simulation, particularly when convergence to the limiting distribution is slow.

Given the difficulties arising with parallelization because of the sequential nature of MCMC simulation, a natural thing to do, discussed by, for instance, Glynn and Heidelberger (1992) and Rosenthal (2000), is simply to generate a separate Markov chain on each processor and combine the results appropriately (see, e.g., Bradford and Thomas 1996). This has the advantage of requiring very little effort beyond that required to program a single-processor version of the MCMC generation code. However, the drawback is that

error associated with burn-in still remains in all processes. Glynn and Heidelberger (1992) and Rosenthal (2000) considered this issue in some detail, and discussed various methods for deciding how much of the initial portion of the chain to remove. (Note also that for low-dimensional problems, it may be preferable to use numerical integration methods, often referred to as "numerical quadrature" or "numerical cubature," instead of MCMC simulation. Such methods are typically trivially parallelizable, but they tend not to perform well in high-dimensional problems.)

In certain problems, the time spent in the burn-in phase may be significant (for instance, if likelihood calculations are long, or if the chain converges very slowly). In such cases it is often desirable to speed up generation of a single chain, rather than use multiple chains. When the state-space of the chain is high-dimensional, one possible way to do this is to divide the state-space into blocks, and then for each iteration of the Markov chain, to handle each block on a separate processor. [This is discussed, for instance, in the nice chapter-length introduction to parallel computing for Bayesian analysis given by Wilkinson (2004), and an interesting example of this kind of approach can be found in Whiley and Wilson (2004).] This approach does indeed speed up generation of a single chain, but requires additional effort in carrying out analysis of the limiting distribution, in order to come up with appropriate blocks. This can be difficult, particularly when the conditional dependence structure in the limiting distribution is complicated. In fact, in certain cases (such as the case study given in this article), it may be impossible. We therefore propose a new algorithm for the purpose of speeding up generation of a single chain by parallelization. The idea is to calculate multiple likelihoods ahead of time ("pre-fetching"), and only use the ones which are needed. The approach does not require any particular analysis of the limiting distribution of the chain (for instance, to subdivide the state-space into blocks). For convenience, we also include in this article brief discussion of two other methods of doing this: the blocking approach mentioned earlier, as well as an approach based on regenerative simulation. We demonstrate the potential gains in performance by applying the pre-fetching method in a time series analysis problem.

The remainder of this article is organized as follows. Section 2 discusses a number of important considerations which arise generally in the context of parallel programming, and briefly discusses existing methods for speeding up generation of single Metropolis-Hastings chains. Section 3 introduces the pre-fetching algorithm, and Section 4 gives results from a study of the performance of the algorithm applied to a particular time series analysis problem. Section 5 concludes with some additional discussion.

2. PARALLEL PROCESSING

2.1 GENERAL CONSIDERATIONS

Conceptually, parallel processing can be applied to almost any problem (*task*) by subdividing it into multiple subtasks. The execution of subtasks may or may not be dependent on the results of other subtasks. When a group of two or more subtasks needs to be executed, and none of the subtasks depends on the results of any of the others, then the subtasks

can be executed concurrently by different processors. Assuming that no single processor is particularly slow relative to the others, this clearly leads to an increase in speed of execution of the original task.

Before considering the specifics of parallel processing for Markov chain Monte Carlo simulation, it is important to keep in mind (at least) the following three factors in this standard approach to parallel processing.

The first is "granularity," that is, the size of the subtasks. In particular, the ratio of the time required to complete a subtask to the time required to carry out interprocess communication is critically important. Typically, machines in a Beowulf cluster communicate via network connections, meaning that even simple messages between processors take on the order of milliseconds to send. On the other hand, the CPUs of the machines can typically execute, for instance, a multiplication operation on the order of picoseconds. Therefore, if the gain in speed due to parallelization is not to be lost because of time spent communicating, it is critical that (relative to communication times) the subtasks each require substantial computational times.

Second, for statisticians using parallel processors to analyze Monte Carlo or Markov chain Monte Carlo problems, it is important to ensure that random number streams on separate processors are independent of each other. Otherwise, one can lose the benefit from parallelizing the Monte Carlo approximation—in the extreme case, if all processors generate exactly the same random number stream, then the result in the parallel approach can be exactly the same as that obtained using a single processor. A simple way of avoiding this problem is to ensure that the random number seed is set differently on each processor, although technically, it is still possible to obtain sequences on separate processors with some degree of overlap. A more sophisticated approach is to use a random number generator specifically designed to generate parallel independent streams. One of these is the `rlecuyer` package for R (http://cran.r-project.org), which is based on algorithms developed by L'Ecuyer, Simard, Chen, and Kelton (2002). Another is the "Scalable Parallel Random Number Generator" library (SPRNG, Mascagni and Srinivasan 2000; see also http://sprng.cs.fsu.edu).

Finally, individual processor speeds may be important. Many Beowulf clusters consist of machines which all run at roughly the same speed. However, in situations where some parts of the cluster are newer than others, it is not uncommon to see a range of different-speed processors. Furthermore, if the cluster is available to many users, then at any given time a machine may be effectively slowed down if one of these users is running a computationally intensive program. As a consequence, a single slow machine may hold up completion of the task. One solution is to employ "load-balancing" to make sure that each processor receives an amount of work proportional to its speed. Two common approaches to doing this are:

1. ensuring that slower processors receive smaller subtasks than faster processors, and
2. adopting a queueing approach.

To use the queueing approach, one simply divides the task into a large number N of small subtasks. "Large" here means any number significantly greater than the number of processors P. Then the first P subtasks can be allocated, one to each processor. The remaining pool

of $N - P$ subtasks is placed in a queue. As soon as the first processor completes its subtask, it is assigned the next subtask, which is removed from the front of the queue. Remaining elements of the queue are subsequently assigned to new processors as the processors become available. This approach ensures that faster processors receive more subtasks, and provides a simple way to carry out automatic load balancing. However, it is effective only when subtasks can be carried out independently and communication time requirements remain small compared to computation time required for the subtasks. Note that when using this approach, if results are to be reproducible when run with identical seeds, it is necessary to ensure that the random number streams are tied to the subtasks, not to the processors.

2.2 EXISTING METHODS FOR GENERATION OF SINGLE CHAINS

The Metropolis-Hastings algorithm and its many variants give us straightforward schemes for obtaining (nonindependent) draws $\{X_1, X_2, \ldots\}$ with the property that as $t \to \infty$, the distribution of X_t converges (in total-variation distance) to a "target distribution" π defined on a state-space Θ (with an associated σ-field). Typically Θ would be m-dimensional Euclidean space $\mathbb{R}^m$, and the target distribution would be the posterior distribution of a set of m real-valued parameters. Our goal is to use parallel processors to speed up simulation of a single Metropolis-Hastings chain with limiting distribution π.

As Wilkinson (2004) pointed out, it may be possible in certain cases to parallelize burdensome likelihood computations directly. For instance, if they involve high-dimensional matrix operations, then the ScaLAPACK library (Choi, Dongarra, Pozo, and Walker 1992; see also http://www.netlib.org/scalapack) may be used to speed up computations. (This technique was used by Whiley and Wilson 2004.) Furthermore, in many cases, it is possible to speed up computation of the target density without resorting to use of parallel processors. Standard techniques for improving program efficiency include working to eliminate unnecessary network/disk accesses and redundant computations, avoiding the use of transcendental functions when possible, and using efficient libraries for optimization, random number generation, and so on. When these approaches still do not provide sufficient improvement in speed, one may still be able to resort to one of the following schemes.

2.2.1 Regeneration

For problems with low-dimensional state-space (i.e., few parameters and latent variables), regeneration (see Mykland, Tierney, and Yu 1995; Brockwell and Kadane 2005 for details) can be used. In the discrete state-space case, use of regeneration to parallelize generation of a single chain is conceptually straightforward. A single point θ^* in the state-space is chosen, and the chain is started from this point. The resulting Markov chain can be divided into segments, each one beginning at θ^* and terminating immediately before the next occurrence of θ^*. These segments, typically referred to as "tours," are independent and identically distributed. Therefore each tour can be generated on a separate processor. The tours can then be patched together in a prespecified order to obtain a single long chain. In

the (more common) continuous state-space case, this approach must be modified slightly, because the probability of a return to the original state θ^* is in most cases equal to zero. However, for low-dimensional state-spaces it is not difficult to adapt the approach. Explicit algorithms were given by Brockwell and Kadane (2005). The primary limitations with this approach, however, are its lack of practical applicability to high-dimensional problems, and the introduction of an additional "re-entry" distribution, which if poorly chosen, can inhibit mixing of the chain.

2.2.2 Blocking

In MCMC problems for which the state-space is high-dimensional, it is tempting to use an update scheme in which within each iteration, each processor is responsible for updating a part of the state-space. However, unless done carefully, this is *not* a valid scheme. The counter-example given in Appendix A illustrates that it does not necessarily yield a Markov chain with the correct invariant distribution. In spite of the fact that this obvious approach does not yield a chain with the correct limiting distribution, under a conditional independence condition which simply requires that the target distribution can be decomposed into the form of a two-level hierarchical model, it is still possible in many cases to make effective use of parallel processing. For convenience, the condition, along with a simple algorithm is stated in Appendix B. [Wilkinson (2004) discussed a more general form of this blocking algorithm, which in a number of cases enables more efficient implementation of parallel block-update schemes, in the sense that one obtains an increase in speed closer to the number of processors P.] For certain problems, however, it may be difficult to establish the conditional independence property required to use the blocking approach, and even when it can be established, the particular block structure may not lead to substantial improvement in speed. In some cases, such as in analysis of long-memory time series models, it may even be impossible to find any conditional independence structure at all.

3. THE PRE-FETCHING ALGORITHM

This article proposes a new method, called *pre-fetching*, for parallel generation of a Markov chain when burn-in time is significant, and the methods mentioned in the previous section are not practical. Suppose that a Metropolis-Hastings chain has already been developed, and that proposal generation and evaluation of prior densities are virtually instantaneous, while the main computational burden is in computing likelihoods. (This is often the case because proposal distributions are usually chosen to have simple density functions and be easy to sample from.)

For the sake of clarity, consider the possible outcomes in only two sequential iterations of the sampler, illustrated in Figure 1. Starting at time t, the chain has state X_t. At time $t+1$, the state X_{t+1} is either the same as X_t (if the proposal is rejected), or is equal to the proposal generated at time t. At time $t+2$, there are four possible outcomes, depending on both the acceptance/rejection in going from time t to $t+1$ and in going from time $t+1$ to

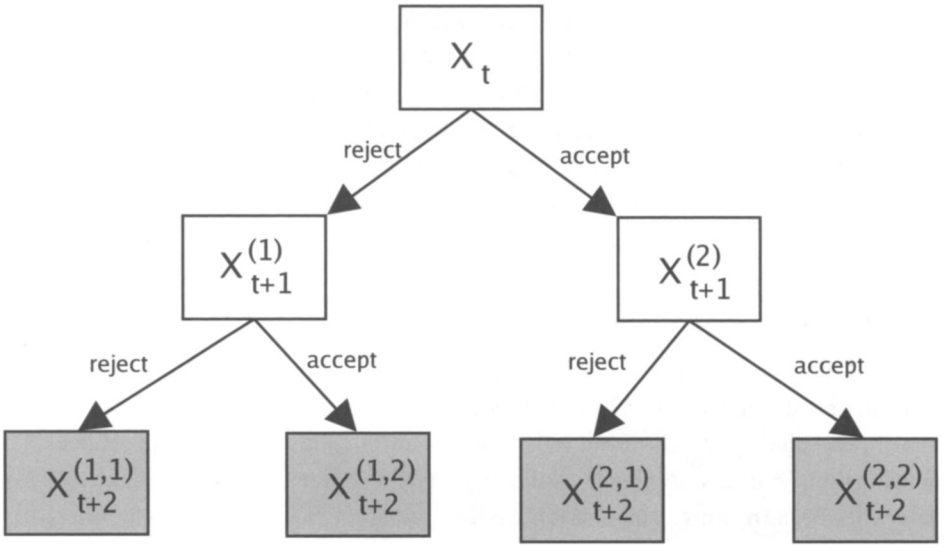

Figure 1. The possible outcomes in two iterations of a Metropolis-Hastings sampler. In two steps, there are only four unique values, those in the shaded leaf nodes, for which likelihoods must be computed. Parents have the same value as their left ("reject")-children.

time $t+2$. In this context, the pre-fetching algorithm assigns one processor to evaluate the likelihood at each one of the four leaf nodes. These likelihoods determine the likelihoods in the parent nodes (a parent node has the same likelihood as its left "rejection" child). Then a single processor gathers the results, and takes two steps down the tree, drawing a uniform random variable each time and comparing it with the appropriate acceptance probability.

Formally, and in the more general case, let

$$X_{t+h}^{(I_1,I_2,\ldots,I_h)}$$

denote the value of the state X_{t+h}, in the case where the proposals in going from time $(t+j-1)$ to $(t+j)$ are

$$\begin{array}{lll} \text{rejected,} & \text{if} & I_j = 1, \\ \text{accepted,} & \text{if} & I_j = 2. \end{array}$$

(Figure 1 shows such values for $h=1$ and $h=2$.) Also, for $j = 0, 1, 2, \ldots, h-1$, let

$$Z_{t+j}^{(I_1,\ldots,I_j)}$$

denote the proposal generated at time $t+j$ (for the value of the state at time $t+j+1$), contingent upon obtaining the corresponding sequence of acceptances/rejections starting at time t. Thus,

$$X_{t+j}^{(I_1,\ldots,I_j)} = \begin{cases} X_{t+j-1}^{(I_1,\ldots,I_{j-1})}, & I_j = 1 \\ Z_{t+j-1}^{(I_1,\ldots,I_{j-1})}, & I_j = 2. \end{cases} \tag{3.1}$$

Suppose also that the possibly time-varying proposal densities are specified by

$$g_t(x_t, z_t) = P(Z_t \in dz_t | X_t = x_t).$$

The pre-fetching algorithm for carrying out h iterations of MCMC simulation, starting with X_t and ending with X_{t+h}, requires 2^h processors. (For good performance, these processors should be balanced, that is, they should be approximately the same speed.)

Algorithm 1: Pre-Fetching

Step 1. Compute all possible proposals and states for the h steps into the future, as follows.

1. Draw the proposal Z_t from the density $g(x_t, z_t)$. Determine $X_{t+1}^{(1)}$ and $X_{t+1}^{(2)}$ using (3.1).
2. Draw proposals $Z_{t+1}^{(1)} \sim g_{t+1}(x_{t+1}^{(1)}, z_{t+1})$ and $Z_{t+1}^{(2)} \sim g_{t+1}(x_{t+1}^{(2)}, z_{t+1})$ and then determine $X_{t+2}^{I_1,I_2}$ for all four possible outcomes $(I_1, I_2) \in \{1,2\}^2$.
3. Repeat this procedure to determine all possible proposals $Z_{t+j}^{(I_1,\ldots,I_j)}$, $(I_1, \ldots, I_j) \in \{1,2\}^j$, $j = 1, 2, \ldots, h$.

Step 2. Identify the 2^h unique possible values that the states $x_t, \ldots, x_{t+h}$ can take. These are simply the values $\{x_t \cup \{x_{t+j}^{(I_1,\ldots,I_j)} : I_j = 2\}, j = 1, \ldots, h\}$. Label these x_i^*, $i = 1, 2, \ldots, 2^h$.

Step 3. Concurrently, on 2^h separate processors, compute the target density $\pi(x_i^*)$, $i = 1, 2, 3, \ldots, 2^h$.

Step 4. For $j = 1, 2, \ldots, h$, compute the realizations i_j of I_j using the standard Metropolis-Hastings rule

$$i_j = \begin{cases} 2, & \text{with probability } \alpha_j, \\ 1, & \text{otherwise,} \end{cases}$$

where

$$\alpha_j = \frac{\pi(z_{t+j}^{(i_1,\ldots,i_{j-1})}) g_t(z_{t+j}^{(i_1,\ldots,i_{j-1})}, x_{t+j}^{(i_1,\ldots,i_{j-1})})}{\pi(x_{t+j}^{(i_1,\ldots,i_{j-1})}) g_t(x_{t+j}^{(i_1,\ldots,i_{j-1})}, z_{t+j}^{(i_1,\ldots,i_{j-1})})}.$$

Step 5. Set $X_{t+j} = x_{t+j}^{(i_1,\ldots,i_j)}$, for $j = 1, 2, \ldots, h$.

Since all computations apart from evaluation of $\pi(\cdot)$ are assumed to be negligible, the speed-limiting component of this algorithm is Step 3. Since likelihoods are computed on separate processors, this algorithm generates h iterations of the Markov chain in the time taken to evaluate the density π only once.

This algorithm is not particularly efficient, because it achieves a speed-increase of only $\log_2(P)$, where P is the number of processors. Essentially, this is because much of the computation is wasted—only one out of 2^h possible paths is actually chosen. However, in cases where none of the previously mentioned methods are practical to implement, it provides a useful alternative. Furthermore, the algorithm is relatively straightforward to implement.

4. SIMULATION STUDY

Fractionally integrated autoregressive moving average models (introduced by Granger and Joyeux 1980; Hosking 1981) have received attention recently as potential improvements over standard autoregressive moving average models in a number of fields, including finance, meteorology, and computer science, where time series appear to exhibit long-range dependence structure.

A zero-mean ARFIMA(p, d, q) process $\{Y_t\}$ satisfies

$$\phi(B)(1-B)^d Y_t = \vartheta(B) Z_t, \tag{4.1}$$

where B denotes the backshift operator, $\phi(B) = 1 - \sum_{i=1}^{p} \phi_i B^i$, $\vartheta(B) = 1 + \sum_{i=1}^{q} \vartheta_i B^i$, the "fractional differencing parameter" d is a constant in the range $(-0.5, 0.5)$, $\{Z_t\}$ is an iid Gaussian noise sequence with mean zero and variance σ^2, and the roots of the polynomials $\phi(\cdot)$ and $\vartheta(\cdot)$ lie strictly outside the unit circle. The fractional differencing operator $(1-B)^d$ is interpreted in the usual manner (see, e.g., Beran 1994) as

$$(1-B)^d = \sum_{k=0}^{\infty} \frac{\Gamma(d+1)}{\Gamma(k+1)\Gamma(d-k+1)} (-1)^k B^k.$$

To facilitate likelihood computations, the process is usually assumed to be Gaussian. When observations $Y_1 = y_1, Y_2 = y_2, \ldots, Y_N = y_n$ are made, the likelihood function is simply that of a multivariate normal,

$$l(y_1, \ldots, y_N; \theta) = (2\pi)^{-N/2} \det(\Lambda)^{-1} \exp(-y^T \Lambda^{-1} y/2),$$

where $y = (y_1, \ldots, y_N)$, and $\Lambda = [\lambda_{ij}]_{i,j=1,\ldots,N}$ denotes the covariance matrix defined by $\lambda_{ij} = \operatorname{cov}(X_i, X_j)$. The components of Λ are computationally tedious to obtain, although the formula given by Sowell (1992) is very helpful. Furthermore, due to the long-memory property of $\{Y_t\}$, the standard (efficient) approach to likelihood evaluation for ARMA models (see, e.g., Brockwell and Davis 1991) cannot be used. Instead, the best existing documented method (to the author's knowledge) is to carry out a Cholesky decomposition of Λ, or to use the Durbin-Levinson algorithm to evaluate $l(y_1, \ldots, y_N; \theta)$. For large values of N (and indeed, one is often interested in the case where N is large when carrying out analysis of long-memory time series), this is time-consuming to compute.

To investigate the performance of the pre-fetching algorithm, we carried out a Bayesian analysis of a simulated ARFIMA(1,d,0) process with parameters $\phi_1 = 0.5, d = 0.3, \sigma^2 = 1$. The simulated process is shown in Figure 2. The state-space of the Markov chain was $\mathbb{R}^3$, with $\{X_t\}$ having limiting distribution equal to the posterior distribution of the parameters ϕ, d, and $\log(\sigma)$. We imposed uninformative (very high variance) priors on parameter ϕ_1, d, and $\log(\sigma)$. Each step in the Metropolis-Hastings algorithm picked one of the three parameters at random, and generated a normally distributed random-walk proposal. Variances of the random-walk step sizes were, respectively, 0.001, 0.0005, and 0.001.

The pre-fetching algorithm was implemented on a Beowulf cluster of 32 dual-CPU 1.6 GHz AMD Athlon Linux systems, connected to each other by 1 gigabit-per-second ethernet

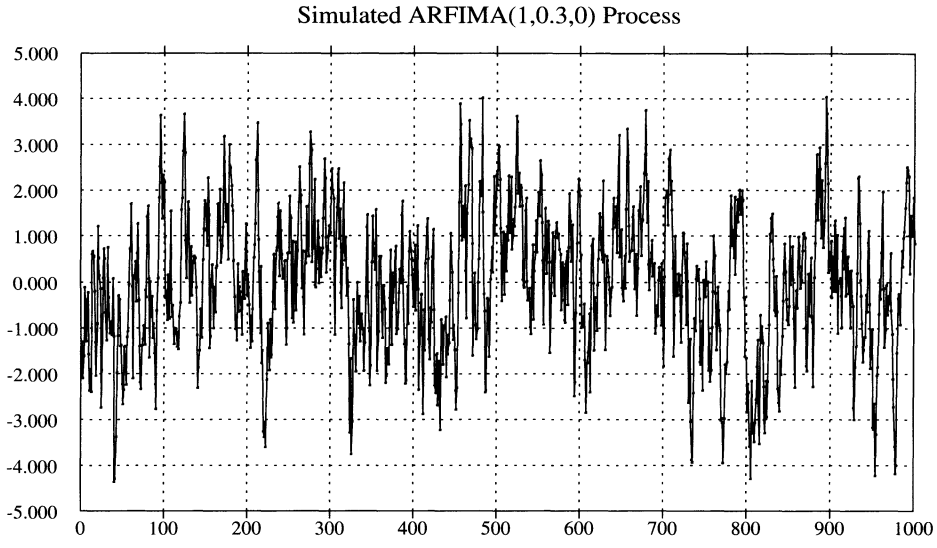

Figure 2. A simulated long-memory (ARFIMA(1,0.3,0)) process of length 1,000.

connections. At the time of running these tests, no other users appeared to be making use of the cluster. Programs were developed in C++, making use of the GNU scientific library, as well as the MPICH implementation (version 1.2.0) of the MPI library. In each run of the pre-fetching version of the MCMC algorithm, 10,000 iterations of a Markov chain were generated, and in all cases, posterior means were indeed close to parameter values used in the simulation. Total execution time was recorded, and used to compute (Markov chain) iterations per second for the scheme. For each chosen number of processors, 60 runs were carried out.

On the machines in this particular cluster, average time to evaluate the likelihood was around eight milliseconds. Boxplots of observed iterations per second, for Markov chains of length 10,000, for $h = 1, 2, \ldots, 5$, are shown in Figure 3. The solid line in the figure indicates the maximum speed we would expect to obtain, based on the assumption that processors are all running at the same speed, there are no "interruptions" on any processors, and that communication between processors is instantaneous, as well as the assumption that actual speed on each processor is the same as the average result obtained in the one-processor runs.

Clearly, actual performance increases roughly as expected when using four processors, but starts to drop away from optimal theoretical performance as one goes to 8, 16, and more processors. This could be explained in part by the inherent sensitivity of the algorithm to variation in individual processing times. In particular, the time taken to carry out one h-step update is the maximum of the times taken to evaluate the likelihoods over all 2^h processors. Thus, a small increase in variance of processing times can have a dramatic effect on overall performance, particularly as h grows. In fact, this would be particularly noticeable if an additional user were to log into one of the 2^h machines and slow it down by

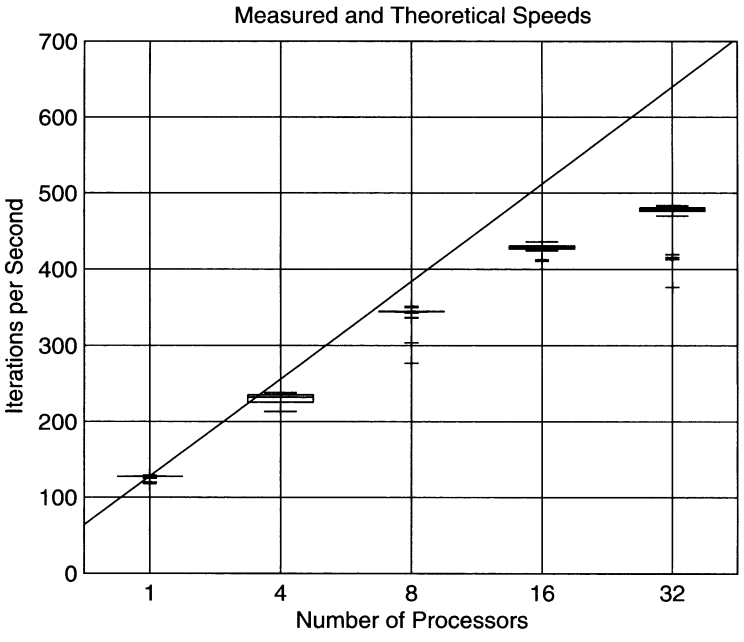

Figure 3. Measured iterations per second for the simulation study, on a 32 dual-CPU Beowulf cluster, as a function of number of processors. Boxplots, each based on 60 runs, are shown for each choice of number of processors. The solid line indicates the theoretical optimal rate.

running additional jobs. In addition, an increased number of processors leads to increased network traffic as more messages are passed to the "master" process. As pointed out in Section 2.1, once the communication time starts to approach the time taken for a single subtask, performance begins to deteriorate.

5. DISCUSSION

This article has introduced a new method for parallelizing generation of a single Markov chain in the context of MCMC simulation. As Wilkinson (2004) emphasized, it is usually worth taking the time to improve mixing of a working single-processor MCMC algorithm, as well as optimizing program speed, before resorting to use of parallel processors. These approaches are typically easier than parallelizing generation of a chain. Furthermore, Rosenthal (2000), Glynn and Heidelberger (1992), and others have also considered the relatively simple approach of generating multiple chains in parallel, which is generally the most efficient solution when burn-in is not a serious problem. The pre-fetching method introduced here is best suited for problems where burn-in time is significant, and other methods are not easy to implement.

The discrepancy between theoretical and actual performance highlighted in the simulation study suggests that, at least when going beyond the simplest four-processor (two-times

speedup) version of this algorithm, further refinements of the algorithm may be useful. By allocating spare processors to the task, there is a range of possible ways to improve this robustness. One approach would be to allocate multiple processors to each likelihood evaluation, and simply take the first value returned. Another more complex approach would be to carry out online evaluation of the variance of response times from processors, and allocate likelihood evaluations preferentially to the "reliable" (i.e. low-variance) machines. Even more complicated versions of the algorithm could be developed that "travel down" the tree as soon as relevant results become available, and then cancel pending likelihood requests which as a consequence become redundant. In addition, it may be worth "streamlining" communications by attempting to reduce the size and/or number of messages sent, in order to reduce the granularity/communication-time problem.

Another intriguing possibility, suggested to the author by an anonymous referee, arises in the case where one can guess whether or not acceptance probabilities will be "high" or "low." In this case, the tree could be made deeper down "high" probability paths and shallower in the "low" probability paths. Theoretically, in such a case, one could exceed the log-base-two of number of processors speedup factor, since there would be a "high" probability of taking a deeper (than $\log_2(P)$) path.

It is also worth noting the potential future benefits of an efficient perfect sampling algorithm, that is, an algorithm which yields a draw from exactly the distribution π. Since the seminal work of Propp and Wilson (1996) appeared, a lot of effort has gone into this area, but so far, no practical general-purpose scheme for typical applied Bayesian analysis problems has been developed. If it were possible to obtain these perfect samples, then the parallel chains approach would clearly be ideal. Each chain would be initialized with an independent perfect sample. This would eliminate the convergence issue, and the chains would then yield independent unbiased parameter estimates.

APPENDIXES

A. AN EXAMPLE OF CARELESS BLOCKING

The following example illustrates how asynchronous updates of separate parts of the state-space may lead to an incorrect invariant distribution for a Metropolis-Hastings chain.

Example A.1. Suppose that the target distribution π is bivariate normal with mean $(0,0)^T$ and covariance matrix

$$\begin{bmatrix} 1 & \rho \\ \rho & 1 \end{bmatrix},$$

and suppose that $X_1 = (X_{11}, X_{12})^T \sim \pi$.

1. The standard Gibbs sampler would generate $X_2 = (X_{21}, X_{22})$ by drawing X_{21} from a $N(\rho X_{12}, 1-\rho^2)$ distribution, and then drawing X_{22} from a $N(\rho X_{21}, 1-\rho^2)$ distribution.

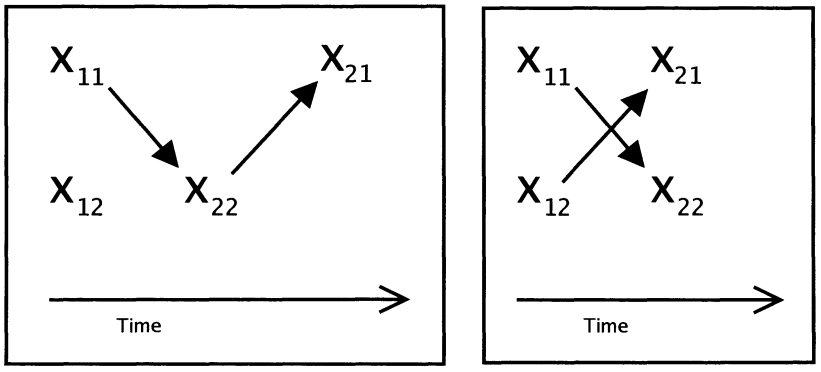

Figure A.1. Left: Standard Gibbs sampling updates for a bivariate target distribution, where one component is updated according to its full-conditional distribution given the other, and then the process is repeated for the other component. Right: Asynchronous sampling updates, where one processor updates each component based on its full-conditional distribution, but updates are carried out concurrently.

This update can be written in the form

$$X_2 = \begin{bmatrix} 0 & \rho \\ 0 & \rho^2 \end{bmatrix} X_1 + \begin{bmatrix} 1 & 0 \\ \rho & 1 \end{bmatrix} \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \end{bmatrix},$$

where ϵ_1, ϵ_2 are iid normal random variables with mean zero and variance $(1 - \rho^2)$. It is easily verified that this gives $X_2 \sim \pi$, and thus π is indeed the invariant distribution of the chain.

2. The asynchronous update sampler would carry out updates of the two components by drawing from their respective full-conditional distributions concurrently. This means X_{21} would be drawn from a $N(\rho X_{12}, 1 - \rho^2)$ distribution, and X_{22} would be drawn from an independent $N(\rho X_{11}, 1 - \rho^2)$ distribution. The update can be written in the form

$$X_2 = \begin{bmatrix} 0 & \rho \\ \rho & 0 \end{bmatrix} X_1 + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \end{bmatrix},$$

where ϵ_1 and ϵ_2 are as defined in the previous case. It is easily checked that this yields

$$X_2 \sim N\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 & \rho^3 \\ \rho^3 & 1 \end{bmatrix} \right).$$

Since the distribution of X_2 is not the same as that of $X_1 \sim \pi$, π cannot be the limiting distribution for the chain.

A simple representation of the state-updates for a bivariate distribution π is given in Figure A.1, with the arrows indicating dependencies in updates.

B. A SIMPLE BLOCKING ALGORITHM FOR GENERATION OF A SINGLE CHAIN

Suppose that the target distribution π can be decomposed into the form of a two-level hierarchical model, that is, that the following property holds.

Property B.1. For some $B \geq 2$, there exists a decomposition of the state-space

$$\Theta = \prod_{i=0}^{B} \Theta_i$$

such that, for all $x = (x^{[0]}, \ldots, x^{[B]}) \in \Theta$, the following equivalent conditions hold.

1. $\pi(x^{[1]}, \ldots, x^{[B]}|x^{[0]}) = \pi(x^{[1]}|x^{[0]})\pi(x^{[2]}|x^{[0]}) \ldots \pi(x^{[B]}|x^{[0]})$.
2. For $j = 1, 2, \ldots, B$, $\pi(x^{[j]}|x^{[-j]}) = \pi(x^{[j]}|x^{[0]})$, where $x^{[-j]}$ denotes the vector $(x^{[0]}, \ldots, x^{[j-1]}, x^{[j+1]}, \ldots, x^{[B]})$.

This can be regarded as a form of the Markov property in which the conditional independence structure is not necessarily determined by time or spatial location. When Property B.1 holds for some state-space Θ and target distribution π, the following parallel Markov chain Monte Carlo algorithm can be used.

Algorithm B.1: Parallel Block-Metropolis-Hastings

Step 1. Choose an initial state $X_1 = (X_1^{[0]}, \ldots, X_1^{[B]})$. Set $k = 1$.

Step 2. Set $X_{k+1} \leftarrow X_k$. Set $k \leftarrow k + 1$.

Step 3. Carry out a Metropolis-Hastings update of $X_k^{[0]}$ (given $X_k^{[1]}, \ldots, X_k^{[B]}$).

Step 4. On B separate processors, carry out concurrent Metropolis-Hastings updates of $X_k^{[j]}$, $j = 1, 2, \ldots, B$. The proposal distribution for updating $X_k^{[j]}$ may depend on $X_k^{[0]}$ as well as the current value of $X_k^{[j]}$ but not on $\{X_k^{[m]},\ m \neq 0, m \neq j\}$.

Step 5. Go back to Step 2.

The scheme here can be regarded as a special case of that described in Wilkinson (2004), where there are two blocks, $T_1 = X^{[0]}$, and $T_2 = (X^{[1]}, \ldots, X^{[B]})$.

Remark: A typical update for $X_k^{[j]}$ in Step 4 would involve drawing a proposal X^* from a distribution $g_j(\cdot; X_k^{[j]}, X_k^{[0]})$, and accepting it with probability

$$\alpha = \min\left(1, \frac{\pi(X_k^{[0]}, X_k^{[1]}, \ldots, X^*, \ldots, X_k^{[B]})g(X_k^{[j]}; X^*, X_k^{[0]})}{\pi(X_k^{[0]}, X_k^{[1]}, \ldots, X_k^{[j]}, \ldots, X_k^{[B]})g(X^*; X_k^{[j]}, X_k^{[0]})}\right).$$

Because Property B.1 is required to hold, this acceptance probability simplifies to

$$\alpha = \min\left(1, \frac{\pi(X^*|X_k^{[0]})g(X_k^{[j]}; X^*, X_k^{[0]})}{\pi(X_k^{[j]}|X_k^{[0]})g(X^*; X_k^{[j]}, X_k^{[0]})}\right).$$

Example B.1: The Generalized State-Space Model

Consider the model

$$\begin{aligned} V_{t+1} &\sim f(\cdot; V_t, \vartheta), \quad t \geq 1 \\ Y_t &\sim g(\cdot; V_t, \vartheta), \end{aligned}$$

where $f(\cdot; V_t, \vartheta)$ and $g(\cdot; V_t, \vartheta)$ are some probability density functions which depend on V_t, as well as a parameter vector ϑ. $\{V_t\}$ is a latent Markov chain, and $\{Y_t\}$ is a sequence of observations whose distributions are determined by $\{V_t\}$. Some assumption is made about the marginal distribution $f(V_1; \vartheta)$—in many cases one can use the stationary distribution of $\{V_t\}$ (assuming it exists) here. In Markov chain Monte Carlo analyses of such models (see, e.g., Carlin, Polson, and Stoffer 1992), one observes $\{y_1, \ldots, y_n\}$, and typically defines the state-space Θ to include all latent variables $\{V_1, \ldots, V_n\}$ as well as the parameter ϑ. For convenience, define $V = \{v_1, \ldots, v_n\}$ and $Y = \{y_1, \ldots, y_n\}$. The posterior distribution is then

$$\pi(\vartheta, V) = kp(\vartheta)p(V|\vartheta)p(Y|V, \vartheta),$$

where k is a normalizing constant which depends on $y_1, \ldots, y_n$, $p(V|\vartheta) = f(v_1; \vartheta) \prod_{t=2}^{n} f(v_t; v_{t-1}, \vartheta)$, and $p(Y|V, \vartheta) = \prod_{t=1}^{n} g(y_t; v_t, \vartheta)$. Now suppose (for the sake of giving an example) that $n = 300$. One could then decompose the state-space into

$$\begin{aligned} \Theta_0 &= (\vartheta, V_{100}, V_{200}), & \Theta_1 &= (V_1, \ldots, V_{99}), \\ \Theta_2 &= (V_{101}, \ldots, V_{199}), & \Theta_3 &= (V_{201}, \ldots, V_{300}). \end{aligned}$$

Under the distribution π, it is relatively straightforward to check that Property B.1 holds. One iteration of the block-Metropolis-Hastings algorithm then involves updating ϑ, V_{100}, and V_{200} (which is generally fairly quick), and then, on three separate processors, concurrently updating the blocks $\{V_1, \ldots, V_{99}\}$, $\{V_{101}, \ldots, V_{199}\}$, and $\{V_{201}, \ldots, V_{299}\}$.

ACKNOWLEDGMENTS

This work was supported in part by NSF Grant IIS-0083418. The author is also grateful to D. Wilkinson, J. Kadane, two anonymous referees, and the editors for their comments and suggestions, and to Brent Frye and the Pittsburgh Brain Imaging Research Center for assistance with use of their Beowulf cluster for the case study considered in this article.

[Received March 2005. Revised June 2005.]

REFERENCES

Beran, J. (1994), *Statistics for Long-Memory Processes*, New York: Chapman and Hall.

Bradford, R., and Thomas, A. (1996), "Markov Chain Monte Carlo Methods for Family Trees using Parallel Processor," *Statistics and Computing*, 6, 67–75.

Brockwell, A. E., and Kadane, J. B. (2005), "Identification of Regeneration Times in MCMC Simulation, with Application to Adaptive Schemes," *Journal of Computational and Graphical Statistics*, 14, 436–458.

Brockwell, P. J., and Davis, R. A. (1991), *Time Series: Theory and Methods* (2nd ed.), Berlin: Springer.

Carlin, B. P., Polson, N. G., and Sroffer, D. S. (1992), "A Monte Carlo Approach to Nonnormal and Nonlinear State-Space Modeling," *Journal of the American Statistical Association*, 87, 493–500.

Choi, J., Dongarra, J., Pozo, R., and Walker, D. (1992), "ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers," in *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, IEEE Computer Society Press, pp. 120–127.

Gilks, W. R., Richardson, S., and Spiegelhalter, D. J. (1996), *Markov Chain Monte Carlo in Practice*, Boca Raton: Chapman and Hall/CRC.

Glynn, P. W., and Heidelberger, P. (1992), "Analysis of Initial Transient State Deletion for Parallel Steady-State Simulations," *SIAM Journal on Scientific and Statistical Computing*, 13, 904–922.

Granger, C. W., and Joyeux, R. (1980), "An Introduction to Long-Memory Time Series Models and Fractional Differencing," *Journal of Time Series Analysis*, 1, 15–29.

Gropp, W., Lusk, E., and Skjellum, A. (1999), *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, Cambridge: The MIT Press.

Hosking, J. R. M. (1981), "Fractional Differencing," *Biometrika*, 68, 165–176.

L'Ecuyer, P., Simard, R., Chen, E. J., and Kelton, W. D. (2002), "An Objected-Oriented Random-Number Package with Many Long Streams and Substreams," *Operations Research*, 50, 1073–1075.

Mascagni, M., and Srinivasan, A. (2000), "SPRNG: A Scalable Library for Pseudorandom Number Generation," *ACMTMS: ACM Transactions on Mathematical Software*, 26.

Mykland, P., Tierney, L., and Yu, B. (1995), "Regeneration in Markov Chain Samplers," *Journal of the American Statistical Association*, 90, 233–241.

Propp, J. G., and Wilson, D. B. (1996), "Exact Sampling with Coupled Markov Chains and Applications to Statistical Mechanics," *Random Structures and Algorithms*, 9, 223–252.

Rosenthal, J. (2000), "Parallel Computing and Monte Carlo Algorithms," *Far East Journal of Theoretical Statistics*, 4, 207–236.

Sowell, F. (1992), "Maximum Likelihood Estimation of Stationary Univariate Fractionally Integrated Time Series Models," *Journal of Econometrics*, 53, 165–188.

Whiley, M., and Wilson, S. P. (2004), "Parallel Algorithms for Markov Chain Monte Carlo Methods in Latent Spatial Gaussian Models," *Statistics and Computing*, 14, 171–179.

Wilkinson, D. J. (2004), "Parallel Bayesian Computation," in *Handbook of Parallel Computing and Statistics*, New York: Marcel Dekker, chapter 18.