Parallel computing and Monte Carlo algorithms

by

Jeffrey S. Rosenthal*

[*Far East Journal of Theoretical Statistics* **4** (2000), 207–236.]

Abstract. We argue that Monte Carlo algorithms are ideally suited to parallel computing, and that "parallel Monte Carlo" should be more widely used. We consider a number of issues that arise, including dealing with slow or unreliable computers. We also discuss the possibilities of parallel Markov chain Monte Carlo. We illustrate our results with actual computer experiments.

Keywords: parallel computing, distributed computing, parallel Monte Carlo, Monte Carlo, Markov chain Monte Carlo, Gibbs sampler, Metropolis-Hastings algorithm, estimation.

1. Introduction.

As computer processors become cheaper and more plentiful, there is great potential for having them compute together in a coordinated fashion. This fact has been known in the computer science community for many years, where the subject of *parallel computing* is very prominent (see for example [29], [16], [30], and [3]), and includes such subspecialities as parallel randomised algorithms (see e.g. [36]) and parallel simulation (see [41] and references therein). A major issue in parallel computing is how to coordinate communication between the various processors; indeed, some parallel computing environments (such as "vector computing") require specialised programming to allow the processors to work together in parallel.

On the other hand, Monte Carlo algorithms (see for example [28], [44]) often proceed by averaging large numbers of computed values. It is sometimes straightforward to have

* Department of Statistics, University of Toronto, Toronto, Ontario, Canada M5S 3G3. Internet: `jeff@math.toronto.edu`. Supported in part by NSERC of Canada.

different processors compute different values, and then use an appropriate (weighted) average of these values to produce a final answer. In this way, the communication between processors is minimised, so that parallel processing is easily facilitated. Indeed, certain Monte Carlo algorithms are so ideally suited to parallel computation that they would be labeled "embarrassingly parallelisable" by computer scientists. However, I do not believe that the Monte Carlo community should be embarrassed about this situation; on the contrary, we should exploit it to the fullest.

Parallel computing can mean many different things, depending on the extent to which the computing is distributed. At one extreme are highly specialised supercomputer hardware arrangements (see e.g. [38]), including Cray supercomputers (see [8]), which require highly specialised software. At the other extreme are World Wide Web based distributed computing efforts (e.g. [10], [12]), which require thousands of volunteers around the world to assist in running software and sending in the results over the internet. Between these two extremes are situations in which a large number of ordinary computers (e.g. in a student computer laboratory, or scattered around the internet) are available for direct control and use (cf. [1], [42], [9]). This situation will likely become more and more common in the future, and this is the situation on which we focus.

Although Monte Carlo is well suited to parallel computation, there are a number of potential problems in the above context. The available computers might run at different speeds; they might have different user loads on them; one or more of them might be down; etc. Handling these issues correctly is crucial to the success of parallel Monte Carlo. In addition, Markov chain Monte Carlo algorithms are now very common (see for example [17], [51], [53], [22], [45]), and parallelising them presents additional difficulties such as determining appropriate burn-in time.

We note that similar issues have been considered in various contexts in the operations research literature. In particular, in an excellent series of papers ([23], [24], [25], [26], [27]), Glynn and Heidelberger have derived many results regarding parallel simulations and weighted averages. In particular, they have considered taking the limit as the number of processors goes to infinity, and have derived central limit theorems, bias expansions, stopping rules, and other information in this context. They have also done some computer experiments [27] related to simulation of queueing networks. In some sense, they have blazed the trail that the statistical Monte Carlo community should follow.

In this paper, we provide a series of "Observations" regarding Monte Carlo computations in a parallel computing environment. Our observations vary from the more practical to the more theoretical, but none should be taken as rigorous mathematical results. Furthermore, most or all of them have undoubtedly been observed elsewhere, by computer scientists and/or operations researchers and/or statisticians. The goal of this paper is to bring such issues more into the mainstream of the Monte Carlo community, and to advocate that parallel Monte Carlo algorithms be more widely used.

This paper is organised as follows. In Section 2, we present the basics of parallel Monte Carlo algorithms. In Section 3, we consider a number of different issues related to possible unreliability of some of the computers being used. In Section 4, we discuss additional issues (especially burn-in time questions) that arise specifically for parallel Markov chain Monte Carlo algorithms. In Section 5, we present some actual parallel Monte Carlo computer simulations. Finally, in Section 6, we provide a few concluding remarks.

2. Parallel Monte Carlo.

Suppose we wish to compute some unknown quantity μ. Suppose further that we have a computer program which is capable of producing an i.i.d. sample $X_1, X_2, \ldots$ of random variables each having mean m and variance v. (Perhaps $m = \mu$; if not, then $m - \mu$ is the bias.)

Standard Monte Carlo would tell us to obtain some large number n of samples, $X_1, \ldots, X_n$, and then to estimate μ by the estimator $E = \frac{1}{n} \sum_{i=1}^{n} X_i$. This estimator would have bias $m - \mu$, and variance v/n.

Now let us consider parallel computing in this context. Suppose we have C computers available to us. We wish to use all of these computers to better estimate μ.

The simplest idea, which could be called parallel i.i.d. Monte Carlo, is to run our same program on each of the C computers. Each computer j then produces n samples $X_1^{(j)}, \ldots X_n^{(j)}$, computes their average $E_j = \frac{1}{n} \sum_{i=1}^{n} X_i^{(j)}$, and reports the result back to a master program. The master program then averages these C results to obtain a master result

$$\overline{E} = \frac{1}{C} \sum_{j=1}^{C} E_j \, . \tag{1}$$

This master result $\overline{E}$ thus has mean m and variance v/Cn. The variance is thus reduced by a factor of C, and our result is equivalent to the result we would have obtained upon running our program on a single computer for C times as long. We have thus obtained linear speed-up. We conclude:

Observation 1. *Simple i.i.d. Monte Carlo algorithms can be run in parallel with little difficulty. The resulting estimator has the same bias, and linearly-reduced variance, compared to the original single-processor estimate.*

We note that a similar observation is made, in considerably greater detail, by Glynn and Heidelberger in [23]–[27].

Remark. In general, running the same algorithm on many different machines requires that the program be copied to, and re-compiled on, each machine. This is not a great difficulty, requiring little time and little thought, and it only has to be done once. (See Section 5 for illustration of such practical matters through actual computer experiments.) Furthermore, if the machines happen to have binary-compatible processors and to have shared memory available to them, then they can share the compiled version of the program and no re-compiling is required. In any case, an appropriate "master program" is required, but this is straightforward; see e.g. [48].

Note in particular that, while we have reduced the variance of our estimate, the bias of our estimate is unchanged (still $m - \mu$ in this case). No amount of parallel computing can reduce this bias. We thus conclude:

Observation 2. *When running parallel Monte Carlo with many computers, it is more important to start with an unbiased (or low-bias) estimate than with a low-variance estimate.*

This fact will guide some of our later choices. Similar observations have been made e.g. in [24].

2.1. Differing computer speeds.

Now suppose that the C available computers are known to run at different speeds (e.g. perhaps some of them are aging IBM 386's, while others are the latest Pentium III's or fast SGI machines). It is then inefficient to obtain the same number n of samples from each machine. Instead, each machine j should be directed to obtain a number n_j of samples roughly proportional to its running speed; and then to compute and report the average $E_j = \frac{1}{n_j} \sum_{i=1}^{n_j} X_i^{(j)}$ of these samples. To make optimal use of these results, the master program should then compute the weighted average

$$\overline{E} \;=\; \frac{\sum_{j=1}^{C} n_j \, E_j}{\sum_{j=1}^{C} n_j} \,. \tag{2}$$

(Note that it is tempting to use the simpler average $\frac{1}{C} \sum_{j=1}^{C} E_j$, but for unequal n_j this simple average will have a higher variance than will $\overline{E}$. This issue is considered in greater detail in [26].) We conclude:

Observation 3. *If using computers of different speeds, they should compute correspondingly different numbers of samples, and their resulting averages should be weighted accordingly.*

Remark. It may be possible to compute the individual sample sizes n_j randomly, for example by having each computer continue until some specified stopping time. However, in this case the sizes n_j may not be independent of the resulting sample values, possibly leading to subtle biases. This is discussed further in Subsection 3.4.

The above outlines the basics of parallel Monte Carlo. It is straightforward to implement, it leads to substantial improvements over ordinary Monte Carlo, and it should be adopted more widely. (See Section 5 for some successful experiments illustrating this.) However, when dealing with a typical collection of real, imperfect computers, certain complications may arise. We consider some of these in the next section.

3. Unreliability issues.

If the C available computers are all single-user, dedicated, and fully maintained, then we can be confident that they will all perform at their usual speed, and report back within an appropriate time. However, if some of the C computers are spread out across the internet, or may be down, or may malfunction, or may have a very high user load, then we cannot be so confident. What is to be done in this case?

3.1. Independent failure events.

Suppose first that we have an *a priori* idea of how long each computation should take, so that we know at what point we should conclude that there is a problem with the i^{th} computer. Let Z_i equal 1 or 0 as the i^{th} computer does or does not report back, correctly, within the *a priori* reasonable amount of time, and let $p_i = \mathbf{P}(Z_i = 0)$ be the probability of failure of computer i.

Suppose first that the variables Z_i are *independent* of the result that the i^{th} computer would have returned. (This is reasonably likely in practice, since most of the possible computer difficulties such as being down, having a high load, etc. are quite separate from the actual computation we ask the computer to do.) Suppose furthermore that the failure probabilities p_i are generally fairly small, so that we expect only a small fraction of the C computers to have problems.

In this case, there is a natural way for the master computer to proceed. Namely, all computers that do not report back correctly within an appropriate amount of time are ignored, and the sample average is computed based only on those computers that succeed. In symbols,

$$\overline{E}_f \;=\; \frac{\sum_{j=1}^{C} n_j\, Z_j\, E_j}{\sum_{j=1}^{C} n_j\, Z_j}\,. \tag{3}$$

Since the failure events are independent of the computed values, such a scheme does not bias the result. And since the failure rate is low, we will obtain only slightly fewer samples than before, so that the variance of our resulting estimate will be only slightly larger. We conclude:

Observation 4. *If there are small probabilities of failures, which are independent of the computed sample values, then it is acceptable for the master program to simply compute the appropriately-weighted average of those results that are returned. This will not bias the resulting estimate, and it will only slightly increase the estimator's variance.*

Remark. Strictly speaking, in the unlikely event that *all* of the computers fail, the estimator $\overline{E}_f$ in (3) would be undefined. Thus, in this and the next subsection, whenever we discuss the bias and variance of $\overline{E}_f$, we are actually referring to $\overline{E}_f$ *conditional* on at least one computer successfully completing its task.

3.2. Dependent failure events.

Suppose now that the variables Z_i are *not* independent of the values that would have been returned. For example, perhaps extremely large sample values tend to crash the computer before the computation finishes. In this case, the resulting estimator $\overline{E}_f$ as in (3) is *not* an unbiased estimator of $m = \mathbf{E}(X_i)$. Rather, each $\mathbf{E}_j$ is an unbiased estimator of $\mathbf{E}(X_j \,|\, Z_j = 1)$, the conditional expected value given that the computer did not fail, which may be somewhat different from the unconditional expected value m.

However, it is possible to control the errors that result from this. Indeed, recalling the definition of *total variation distance* $\|\nu - \rho\|$ between two probability measures ν and ρ, namely

$$\|\nu - \rho\| \;=\; \sup_A |\nu(A) - \rho(A)| \;=\; \sup_{0 \le f \le 1} \left| \int f\, d\nu - \int f\, d\rho \right| , \tag{4}$$

we claim that

$$\|\mathcal{L}(X_i \,|\, Z_i = 1) - \mathcal{L}(X_i)\| \;\le\; \mathbf{P}(Z_i = 0)\,.$$

(Here $\mathcal{L}$ stands for *law* or *distribution*; thus, $\mathcal{L}(X_i \,|\, Z_i = 1)$ is the conditional distribution of X_i, given that $Z_i = 1$.)

To prove this, note that

$$\|\mathcal{L}(X_i \,|\, Z_i = 1) - \mathcal{L}(X_i)\| \;=\; \sup_A \Big| \mathbf{P}(X_i \in A \,|\, Z_i = 1) - \mathbf{P}(X_i \in A) \Big|$$

$$= \; \sup_A \Big| \mathbf{P}(X_i \in A \,|\, Z_i = 1) - \mathbf{P}(X_i \in A \,|Z_i = 0)\mathbf{P}(Z_i = 0) - \mathbf{P}(X_i \in A \,|Z_i = 1)\mathbf{P}(Z_i = 1) \Big|$$

$$= \sup_A \left| \mathbf{P}(X_i \in A \mid Z_i = 1)\,(1 - \mathbf{P}(Z_i = 1)) - \mathbf{P}(X_i \in A \mid Z_i = 0)\mathbf{P}(Z_i = 0) \right| . \qquad (5)$$

But $(1 - \mathbf{P}(Z_i = 1)) = \mathbf{P}(Z_i = 0)$, so that the difference in (5) is the difference of two non-negative terms, each of which is $\leq \mathbf{P}(Z_i = 0)$. Hence, their difference is also $\leq \mathbf{P}(Z_i = 0)$, and the result follows.

It now follows from the definition (4) that, if (say) $0 \leq X_i \leq M$, then $\mathbf{E}(X_i) - \mathbf{E}(X_i \mid Z_i = 1) \leq M\,\mathbf{P}(Z_i = 0)$. Furthermore, we recall from (3) that $\overline{E}$ is a weighted average of various unbiased estimates of $\mathbf{E}(X_i \mid Z_i = 1)$.

We therefore conclude:

Observation 5. *If the computer failure events are not independent of the resulting sample values, but if their probabilities are bounded above by p, and if the possible sample values are restricted to a bounded interval $[0, M]$, then the bias of the resulting estimator $\overline{E}_f$ is at most Mp.*

3.3. Repeating failed experiments.

There is an alternative approach to dealing with dependent computer failures as above. (By Observation 5, this might be most useful when the failure probabilities are not small and/or the sampled values are not bounded.) Namely, we can decide to *repeat* (on a different computer) any computation which did not report back within an appropriate amount of time.

In general it is non-trivial to do this; the computations involve *randomness* and hence cannot necessarily be repeated exactly. Furthermore it does not suffice to do an independent replication of the random experiment, since this does not eliminate the bias from dependent computer failures as in the previous subsection.

On the other hand, the random computations typically involve only *pseudo-randomness*, i.e. numbers obtained from a pseudo-random number generator. If the computation is started again with the identical pseudo-random *seed*, then the pseudo-random number generator will produce identical pseudo-random numbers, and hence will exactly duplicate the desired computation.

This suggests a new way to handle computer failures. We let the master program assign the pseudo-random number seeds to each computer, say $s_1, \ldots, s_C$. Once a computer (say, computer j) finishes its computation and reports back, it is then assigned a fresh computation with a seed s_i taken from some computer that has not yet reported back. (If all computers that have not yet reported back *already have* a second computer using the same seed, then we would start a *third*, or fourth or fifth, computer from the same seed.) In this way, eventually we obtain (at least once) a result based on each of the C different seeds (even though some computers may not have reported back at all, and other computers may have done several different computations). We can then average these C results to obtain an estimator $\overline{E}_r$. The value of $\overline{E}_r$ depends only on the initial choice $s_1, \ldots, s_C$ of seeds, and is unaffected by any computer failures along the way.

On the other hand, if there is at least one failure, then it will take *longer* to compute $\overline{E}_r$ than it would have to compute $\overline{E}_f$ as in the previous subsection. Indeed, assuming that between 1 and $C/2$ of the computers fail, and that each computation takes approximately the same amount of time (assuming it does not fail), then to compute $\overline{E}_r$ will require some computers to do *two* computations, but no computer to do more than two computations. The time to compute $\overline{E}_r$ will therefore be about *twice* as long as the time to compute $\overline{E}_f$. We conclude:

Observation 6. *An alternative approach to computer failure is to have all pseudo-random number seeds assigned by the master program, and then to re-compute any failed computations, using the same seed on a fresh computer. This approach results in an estimator whose value is unaffected by computer failure. The total computation time is increased by approximately a factor of 2.*

Remarks. Even if we do not plan to repeat any experiments, it still may be a good idea for the master program to provide pseudo-random seeds to all the computers. Indeed, if instead each computer chooses a seed based on the current clock time, then there is some risk that two computers will accidentally pick the same seed. We also note that it is possible, due to implementation details such as floating-point rounding methods, that two different computers may not get the same answer *even if* running the identical algorithm with identical pseudo-random seed. It is even conceivable that such differences could bias the resulting estimates. However, this seems unlikely and we do not consider it further here.

3.4. Variable or unknown computer speeds: choosing n_j on-line.

As mentioned in Subsection 2.1, with differing computer speeds it is sometimes necessary to choose different values n_j (corresponding to different numbers of simulated values to be generated by computer j), with n_j proportional to the computing speed of the j^{th} computer. The resulting estimates are then averaged together according to a weighted average as in (2).

However, in some cases we will not know in advance the relative speeds of the different computers involved. Indeed, because of heavy user loads, it may be difficult or impossible to predict in advance the computational speed of certain computers. Thus, it is desirable to have a way of determining the values n_j on-line.

A natural way to proceed is to specify in advance the total amount of time T that we want each computation to take. We can then instruct the various computers to keep on simulating until time T, at which point they should simply average the values they have obtained so far, and report that result.

A question that arises is what the computer should do if a simulation is *in progress* at time T. This might seem like an unimportant detail. However, if the value of a simulation is dependent upon the time taken to compute the simulation (which will often be the case), then it is easy to see that we may bias our result if we do not handle this situation properly. And, in light of Observation 2, we wish to avoid bias as much as possible.

Such issues were investigated by Glynn and Heidelberger [23] (see also [24], [25], [26], [15], [(??)], and Remark 5.3 of [14]), where it was shown that to avoid biasing the result, the following *Unbiased Stopping Rule* should be used: *When time T approaches, the simulation in progress should be continued if and only if it is still the first simulation; otherwise, the simulation in progress should be discarded.* If the computers follow this rule, then their resulting estimate will be unbiased. However, if instead they always allow the simulation in progress to complete, then they will in general introduce a bias of order $1/T$ into the result.

Hence, we conclude:

Observation 7. *When using computers of unknown or highly variable speed, we may determine the sample numbers n_j on-line, by pre-specifying a total computation time T and then following the Unbiased Stopping Rule as above. Such a procedure will not introduce any additional bias into our result, and will allow us to make use of whatever computer power is available to us.*

Of course, if the time per simulation is extremely small, or is not heavily dependent on the value computed, then it is less important how we treat the simulation in progress, and it might not be necessary to bother with the Unbiased Stopping Rule. See [23] for further discussion and analysis.

Remark. It may be difficult to combine this "determine n_j on-line" approach with the "repeated experiment" approach presented in Subsection 3.3. Indeed, if the number of simulations n_j is allowed to depend on the speed at which the simulation happens to run, then it follows that a second run would not produce identical results even if started with the same pseudo-random number seed.

4. Parallel Markov chain Monte Carlo.

Markov chain Monte Carlo (MCMC) algorithms (see e.g. [17], [51], [53], [22], [45]), such as the Gibbs sampler and the Metropolis-Hastings algorithm, have become extremely popular in statistics (especially Bayesian statistics) as a method of approximately computing difficult high-dimensional integrals. They are also used in theoretical computer science for approximate counting problems (see e.g. [50]). Like classical Monte Carlo, these algorithms proceed by generating a sample $X_1, X_2, \ldots$. However, in this case the sample values are generally functionals of a *Markov chain* (i.e., $X_i = g(Z_i)$ where $\{Z_i\}_{i=0}^{\infty}$ is a Markov chain, with Z_0 chosen from some appropriate initial distribution) and are thus *dependent*. For large i, the distribution of X_i is approximately stationary, with mean equal to μ, the quantity to be estimated. However, for smaller i this may not be the case.

Remark. Of course, one might well be interested in estimating several functionals of the Markov chain, say $X_i = g(Z_i)$ and $Y_i = h(Z_i)$. In this case, all of our comments would apply equally well to $\{X_i\}$ and to $\{Y_i\}$ together. In this paper, we consider a single functional solely for ease of exposition.

A typical estimator from an MCMC algorithm is of the form $\frac{1}{N-B+1}\sum_{i=B}^{N} X_i$, i.e. $\frac{1}{N-B+1}\sum_{i=B}^{N} g(Z_i)$. Here B is a "burn-in time", designed to be large enough that the distributions of $X_B, X_{B+1}, \ldots$ are approximately stationary. However, it is often not obvious how to choose a value for B, and indeed this is one of the biggest difficulties with MCMC algorithms.

For parallel Markov chain Monte Carlo, we can have each computer j generate n_j samples $X_1^{(j)}, \ldots, X_{n_j}^{(j)}$, and compute and return an estimate $E_j = \frac{1}{n_j - B_j + 1}\sum_{i=B_j}^{n_j} X_i^{(j)}$. As before, the master program can then compute an overall estimate

$$\overline{E} \;=\; \frac{\sum_{j=1}^{C} n_j\, E_j}{\sum_{j=1}^{C} n_j}$$

as in (2). We thus record that:

Observation 8. *Like i.i.d. Monte Carlo algorithms, Markov chain Monte Carlo algorithms can also be easily run in parallel.*

We note that any burn-in period must be run separately on each computer, so that the resulting speed-up is slightly less than linear in this case.

Of course, all of the unreliability problems and solutions for ordinary parallel Monte Carlo, discussed in the previous section, still apply. Furthermore, if the Markov chain is slowly mixing then the resulting samples $X_1^{(j)}, \ldots, X_{n_j}^{(j)}$ may be highly correlated, thereby increasing the variance of $\overline{E}$ somewhat; but this is an unavoidable consequence of using MCMC of any kind.

One new issue which arises for parallel MCMC, but not for parallel i.i.d. Monte Carlo, is the choice of the burn-in times B_j. It is sometimes suggested (e.g. [20]) that, as long as the total number of samples n_j is large enough, the burn-in time isn't so important. [Indeed, as $n_j \to \infty$, the burn-in bias will only effect the resulting estimate by $O(1/n_j)$, while random sample variability will effect the resulting estimate by $O(1/\sqrt{n_j})$.]

However, for parallel MCMC, the total computation is divided into C different pieces, so we may not have each n_j as large as for conventional MCMC. Hence, if C is large, then burn-in issues are still very important. (More formally, the burn-in bias will still be $O(1/n_j)$, while random sample variability will now be $O(1/\sqrt{Cn_j})$.) By Observation 2, for parallel MCMC we want to eliminated bias as much as possible. (Similar issues were considered, both experimentally and theoretically as $C \to \infty$, by Glynn and Heidelberger [24], [26], [27].) We now discuss different methods of choosing B_j, with Observation 2 kept in mind.

4.1. Fixed burn-in times.

One possibility is to simply choose $B_j = K$, for some fixed pre-chosen constant K. (Perhaps $K = 0$, or perhaps K is a function of the number of processors C.) This is easy to implement, and if n_j is large enough, the resulting estimates E_j may be reasonably close to μ. However, if K is too small, e.g. if the corresponding Markov chain is "slowly mixing", then the estimators E_j may be substantially biased. Similarly, if K is much too large, then lots of computational effort is wasted. Furthermore, in the absence of additional information about the Markov chain under consideration, it is usually very difficult to determine an appropriate constant K in advance. Hence, this method of choosing $B_j = K$ would appear to have limited appeal for parallel MCMC in general.

4.2. Convergence diagnostics.

In the absence of good theoretical knowledge of appropriate burn-in times B_j, it is common to use *convergence diagnostics* (see e.g. [18], [5], [2]) to determine the burn-in time. Here the values B_j are chosen on-line, based on statistical analysis of the sample run $X_1^{(j)}, X_2^{(j)}, \ldots$ (or perhaps of the underlying Markov chain run $Z_1^{(j)}, Z_2^{(j)}, \ldots$) in progress.

Such convergence diagnostics often work well in practice. However, they have at least two draw-backs from the parallel MCMC perspective. First, they sometimes prematurely diagnose convergence by providing a burn-in time B which is too small (see e.g. [33], [5]), leading to biases as in Subsection 4.1 above. Second, as shown in [6], by basing the burn-in time on the sample in progress, convergence diagnostics sometimes introduce biases of their own (even if the Markov chain converges immediately).

We thus record:

Observation 9. *When running parallel Markov chain Monte Carlo, choice of burn-in is a very important issue. Convergence diagnostics or fixed-length burn-in can be used, but they may introduce bias into the resulting estimate.*

Remark. It is possible to use the parallel MCMC results themselves as a form of diagnostic. Specifically, if the estimates E_j from the different computers are all very different, then this may suggest that the Markov chain has not yet converged, or that there is a problem with the algorithm. (This is somewhat related to the multiple-runs computer diagnostics of e.g. [18].) However, such "diagnostic" method should be used with care, to avoid introducing additional biases (cf. [6]). In addition, it may be possible to do more sophisticated analysis (e.g. of autocorrelations) on the multiple parallel runs, though this may require greater communication between the different computers.

4.3. Theoretical quantitative bounds.

It is sometimes possible (cf. [34], [46], [47]) to use theoretical analysis to compute a burn-in time B such that the distribution of the Markov chain after B steps is provably within ϵ of its stationary distribution. Such theoretical analysis is too difficult to be used routinely, however it has been successfully applied to some reasonably complicated examples of the Gibbs sampler (see e.g. [47]). Furthermore, certain auxiliary-simulation methods have been proposed ([7], [4]) to estimate such theoretical burn-in times B in more general situations.

If we are able to obtain (or estimate) such a theoretical burn-in time B, then implementing parallel MCMC is straightforward. Indeed, we simply set $B_j = B$ for each computer j, and then compute E_j and $\overline{E}$ as above. The resulting estimator $\overline{E}$ has extremely small bias ($\leq \epsilon M$ if the possible values of X_i are in the interval $[0, M]$), and makes good use of the computing power of all the available computers. We conclude:

Observation 10. *When running parallel Markov chain Monte Carlo, if theoretical burn-in bounds are available (or can be estimated), then they should be used to reduce bias in the resulting estimates.*

4.4. Perfect simulation.

A recent exciting development in MCMC methodology is the establishment of *perfect simulation algorithms* by Propp and Wilson [43] and Fill [14] (see also [54] and references therein). These algorithms use a Markov chain in a clever way, to produce a random variable Z_0 which is distributed *exactly* according to its stationary distribution. It then follows that $X_0 = g(Z_0)$ has precisely its stationary distribution, and in particular we have $\mathbf{E}(X_0) = \mu$. Such algorithms are therefore completely unbiased, and thus very attractive for parallel MCMC purposes. Similarly, it is sometimes possible to produce perfect samples by non-Markovian methods, e.g. using rejection sampling.

One difficulty is that such perfect samples (when they can be generated at all) often take a rather long time to generate. Thus, it may be very inefficient to simply repeat the perfect simulation algorithm over and over, and thus generate perfect i.i.d. stationary samples. A natural idea (analysed in [37]) is to generate Z_0 using a perfect simulation algorithm, and then generate $Z_1, Z_2, \ldots$ simply using the Markov chain updating rule. In this case, setting $X_i = g(Z_i)$, the resulting samples $X_0, X_1, \ldots$ will be highly dependent in general. However, each sample will individually be distributed according to the stationary distribution. In particular, for each i we will have $\mathbf{E}(X_i) = \mu$. Thus, all the bias (inherent in ordinary MCMC) has been eliminated.

This idea carries over to parallel MCMC. If a perfect simulation algorithm exists (even if it is fairly slow), then we can proceed as follows. We instruct each computer j to begin by computing a single perfect sample $Z_0^{(j)}$. The computer then uses this value as the initial value in an ordinary Markov chain run $Z_0^{(j)}, Z_1^{(j)}, \ldots, Z_{n_j-1}^{(j)}$. When finished, the computer returns the estimate $E_j = \frac{1}{n_j} \sum_{i=0}^{n_j-1} X_i^{(j)} = \frac{1}{n_j} \sum_{i=0}^{n_j-1} g(Z_i^{(j)})$. (Note that now no burn-in time is required.) The master program then computes an overall estimate $\overline{E}$ as in (2).

Such a scheme is completely unbiased. Furthermore, each computer only has to compute a (slow) perfect sample once, and thereafter just computes (fast) Markov chain updates. Thus, each computer makes good use of its available computational power. We conclude:

Observation 11. *When running parallel Markov chain Monte Carlo, if perfect samples are available, then they should be used once (even if they are slow) on each computer, to initialise the Markov chain exactly in its stationary distribution, and thereby eliminate all initialisation bias.*

4.5. Stopping rule.

If the computers are running Markov chains, then their simulated values are *not* exchangable like they were in the i.i.d. case. In particular, the Unbiased Stopping Rule (as in Observation 7) no longer guarantees an unbiased estimate.

Despite this, the Unbiased Stopping Rule still provides a good stopping rule for simulating from Markov chains. This seems intuitively clear by analogy to the i.i.d. case. It is also implied by the more careful analysis of Glynn and Heidelberger ([26], Theorem 9), who derive asymptotic expansions for the bias of this rule as the number of processors goes to infinity.

Thus, we continue to use the Unbiased Stopping Rule in the MCMC case, even though it may no longer be a perfectly unbiased stopping rule.

4.6. Multiple Markov chain Monte Carlo.

There is a more specialised Monte Carlo algorithm which has been suggested as a candidate for parallelisation. This is the *Metropolis-coupled Markov chain Monte Carlo* (or, *multiple Markov chain Monte Carlo*) method of Geyer [19]; see also [21], [52], [40].

The algorithm is related to simulated tempering (see [32], [21], [31]) and to tempered transitions (see [39]). It proceeds by simultaneously running a number m of different Markov chains, governed by different (but related) Markov chain transition probabilities. It occasionally "swaps" values from two different chains, with probabilities governed by the Metropolis algorithm to preserve stationarity of the target distribution. These swaps hopefully speed up convergence of the algorithm, perhaps very substantially.

Now, if the number m of different Markov chains to be run happens to be the same as the number C of computers available, then it is natural to try to parallelise this algorithm. Specifically, each of the m different Markov chains could be run simultaneously on a different computer, with the computers occasionally communicating for purposes of engineering a swap.

If each computer is running perfectly reliably, and at the same rate, and if they are able to communicate efficiently, then no difficulties arise, and we conclude:

Observation 12. *If we have an ideal computing environment, in which m reliable computers are available, run at the same rate, and communicate without difficulty, then Metropolis-coupled Markov chain Monte Carlo algorithms can be effectively run in parallel.*

However, in a non-ideal environment, parallel Metropolis-coupled Markov chain Monte Carlo presents a number of additional difficulties to be overcome, such as:

1. How do the different computers communicate with each other in order to initiate "swaps" in an efficient manner?
2. What do we do if one of the computers is down or fails to respond?
3. If the computers are running at different speeds, is it acceptable to swap values from two Markov chains that have completed a *different number* of iterations?
4. If yes to question three, then if the number of iterations completed depends on the values computed, will this introduce bias into the resulting estimate?
5. What if $C < m$, i.e. we have fewer computers available than different chains we wish to run? Is it worth reducing the number m of chains in order to fit the available hardware?
6. What if $C > m$, i.e. we have more computers available than different chains we wish to run? Is it worth increasing the number m of chains to make best use of the available hardware?

We leave these as open questions for future research. Their investigation would appear to be important for effective implementation of parallel Metropolis-coupled Markov chain Monte Carlo in non-ideal computing environments.

5. Computer experiments.

To better illustrate the parallel Monte Carlo algorithms we are advocating, we have performed three very simple computer experiments. (For related experiments in the context of queueing systems, see [27].)

For each of the experiments, we have run parallel simulation algorithms on five computers to which the author happened to have access. (Of course, in a more serious use of parallel Monte Carlo, the number of computers would be much greater than five, perhaps numbering in the hundreds or thousands.) These five computers are all connected by the internet (one of them at a distance of 100 kilometers). All five run some version of the Unix operating system, and thus can easily run computer programs written in C.

To carry out the experiment, a master program `jpar.c` [48] was used. This program used the `rsh` command to remotely run the compiled C programs on each of the five computers, and used the `popen` command to collect the results. These results were then averaged together, using weightings equal to the number of simulations represented by each result.

The individual computers were each instructed to terminate their simulation after 60 seconds. Furthermore, the Unbiased Stopping Rule (cf. Observation 7) was used on each computer to discard the iteration in progress when the 60 second deadline approached.

Each of the five computers are shared by many users; thus, the programs were all executed using the `nice` command to reduce the amount of CPU time they took away from other programs.

5.1. Uniform variable averaging.

As a simple first example, we considered the case where the X_i are all i.i.d. uniformly distributed on $[0,1]$, and we are simply interested in computing their average. This case does not involve Markov chains or any other such complications. The simple program `unifavr.c` [48] was compiled on each computer (using a simple shell-script). It was then simultaneously run on each computer (using the master program `jpar.c`), generating and averaging Uniform$[0,1]$ pseudo-random numbers. The program on each computer was instructed to cease computation (and issue a report to the master program) after 60 seconds. The results are summarised in Figure 1.

Computer	Running Time	# Samples	Estimate
1. Linux 2.0.35	60.11064 sec	15,177,767	0.499937
2. Irix 6.5	60.43933 sec	10,747,492	0.500224
3. Irix 5.3	60.76017 sec	2,427,713	0.500008
4. Irix 6.2	60.52251 sec	4,925,341	0.499848
5. Linux 2.0.27	60.57984 sec	6,739,259	0.499954
MASTER	60.76256 sec	40,017,572	0.50001029

Figure 1. Results of the `unifavr` simulation.

As can be seen, the total time used by each computer was just slightly larger than the 60 second target time. Furthermore, the total time for the entire parallel experiment was essentially no larger than this; in other words, there was virtually no wasted time in coordinating and collating the individual computers' outputs. On the other hand, the total number of simulations averaged into the final "master" estimate is the *sum* of the individual simulation totals. This justifies the claim (Observation 1) that i.i.d. parallel Monte Carlo can be run easily, and gives essentially linear speed-up over a single-computer simulation.

The master program then produced a final estimate, based on the weighted sum (2), making use of all the simulations on all five computers. The resulting master estimate is, of course, extremely close to its expected value of 0.5.

5.2. A one-dimensional Markov chain.

We next consider a very simple Markov chain example. Here the Markov chain is given by $Z_0, Z_1, \ldots$, where $\mathcal{L}(Z_{i+1} \mid Z_i) = N(Z_i/2,\ 3/4)$. That is, given a value of Z_i, we choose Z_{i+1} from a normal distribution with mean $Z_i/2$ and variance 0.75.

For illustrative purposes, let us suppose we are interested in estimating the expected value $\mathbf{E}[(Z_i)^2]$, i.e. the expected value $\mathbf{E}[g(Z_i)]$ where $g(z) = z^2$. We therefore set $X_i =$

$g(Z_i) = (Z_i)^2$, and use the estimator

$$E_j = \frac{1}{n_j} \sum_{i=1}^{n_j} X_i$$

on the j^{th} computer.

Now, it is well-known (see e.g. [49], [46]) that this Markov chain has as its stationary distribution the standard normal distribution $N(0, 1)$. For illustrative purposes, we pretend that it is difficult and time-consuming to sample directly from this distribution. We thus proceed (as in Observation 11 above) by having each computer j begin with a single exact $N(0, 1)$ sample, $Z_0^{(j)}$, and then run the Markov chain to obtain further samples $Z_1^{(j)}, Z_2^{(j)}, \ldots$. (If, instead, the chains began with a large fixed value like $Z_0^{(j)} = 1000$, then this would serious bias the resulting estimates unless n_j were extremely large.)

Since the chains begin with an exact sample, they do not require any burn-in period, so the j^{th} computer uses the estimator $E_j = (1/n_j) \sum_{i=1}^{n_j} X_i$ as above. Here, like in the previous example, n_j is the number of simulations that can be computed within the target 60 second time-frame.

The experiment was carried out, again using `jpar.c` as the master program, and this time using `normalmc.c` [48] on each individual computer. The results are summarised in Figure 2.

Computer	Running Time	# Samples	Estimate
1. Linux 2.0.35	60.09057 sec	5,471,868	1.000091
2. Irix 6.5	60.34377 sec	4,919,156	1.000248
3. Irix 5.3	60.81072 sec	915,644	1.001021
4. Irix 6.2	60.56002 sec	2,174,534	1.003149
5. Linux 2.0.27	60.93772 sec	2,242,576	0.999483
MASTER	60.94014 sec	15,723,778	1.00053044

Figure 2. Results of the `normalmc` simulation.

Once again, the parallel algorithm proceeded efficiently and gave linear speed-up (cf. Observation 1). Furthermore, because each computer began with an exact sample, there was no initialisation or burn-in bias (cf. Observation 11).

And, once again, the results are very close to the expected value, which in this case is $\mathbf{E}(X_i) = \mathbf{E}((Z_i)^2) = 1.0$.

5.3. A random-effects Gibbs sampler Markov chain.

For our final example, we consider a more complicated Markov chain. This chain, described in [47], arises from a Gibbs sampler related to a model for James-Stein estimates applied to baseball players' batting averages (see Efron and Morris [13], Morris [35]).

This Markov chain corresponds to the following model. For $1 \leq i \leq K$, we observe data Y_i, where $Y_i \,|\, \theta_i \sim N(\theta_i, V)$ and are conditionally independent. Here θ_i are unknown parameters to be estimated, and $V > 0$ is assumed to be known. Furthermore $\theta_i \,|\, \mu, A \sim N(\mu, A)$ and are conditionally independent. Finally, μ has a flat prior, and A has a prior of the form $IG(a, b)$, where IG is the inverse gamma distribution with density proportional to $e^{-b/x} x^{-(a+1)}$, and where a and b are fixed constants.

The Markov chain runs on the random variables $Z = (A, \mu, \theta_1, \ldots, \theta_K)$. We index them by a time parameter k, so that the Markov chain variables are $Z_k = (A_k, \mu_k, \theta_{k,1}, \ldots, \theta_{k,K})$, for times $k = 0, 1, 2, \ldots$. Given initial values $\theta_{0,1}, \ldots, \theta_{0,K}$, the chain proceeds by, for $k = 1, 2, \ldots$, repeatedly sampling

$$A_k \;\sim\; \mathcal{L}(A \,|\, \theta_1 = \theta_{k-1,1}, \ldots, \theta_K = \theta_{k-1,K}, Y_1, \ldots, Y_K)$$

$$= \; IG\left(a + \frac{K-1}{2},\; b + \frac{1}{2}\sum_{i=1}^{K}(\theta_{k-1,i} - \overline{\theta}_{k-1})^2\right) \;;$$

$$\mu_k \;\sim\; \mathcal{L}(\mu \,|\, A = A_k, \theta_1 = \theta_{k-1,1}, \ldots, \theta_K = \theta_{k-1,K}, Y_1, \ldots, Y_K) \;=\; N(\overline{\theta}_{k-1},\; A_k/K)\,;$$

and then for $i = 1, 2, \ldots, K$,

$$\theta_{k,i} \;\sim\; \mathcal{L}(\theta_i \,|\, A = A_k, \mu = \mu_k, Y_1, \ldots, Y_K) \;=\; N\left(\frac{\mu_k V + Y_i A_k}{V + A_k},\; \frac{A_k V}{V + A_k}\right) \;;$$

here $\overline{\theta}_{k-1} = \frac{1}{K}\sum_{i=1}^{K} \theta_{k-1,i}$.

[This Markov chain corresponds to a Gibbs sampler on the $(K+2)$ variables $(A, \mu, \theta_1, \ldots, \theta_K)$, where the pair (A, μ) is treated as a single block; for further details see [47].]

We consider this chain with $K = 18$, $V = 0.00434$, $a = -1$, $b = 2$, and the data $Y_1, \ldots, Y_{18}$ as in Table 1 of Morris [35]. Furthermore, we consider initial values $\theta_{0,1} = \ldots = \theta_{0,18} = \overline{Y}$, where $\overline{Y} = \frac{1}{18}\sum_{i=1}^{18} Y_i$.

For these values, it was rigorously proved in [47] that the Markov chain will converge (to within 1% of its stationary distribution) after at most 140 iterations. Hence, as in Observation 10, we should instruct each computer, when running the Markov chain, to *discard* all iterations before the 140^{th}, and base their estimate solely on iterations $140, 141, \ldots, n_j$.

An experiment was carried out, again using `jpar.c` as the master program, and this time using the program `james.c` [48] (which in turn uses random-number generation algorithms presented in [11]) on each individual computer. For illustrative purposes, we have concentrated on estimating the expected value of the θ_1 variable (i.e. we have set $X_k = g(Z_k) = \theta_{k,1}$). Each computer was again instructed to iteratively run the Markov chain, terminating after 60 seconds; once again, the Unbiased Stopping Rule (cf. Observation 7) was used on each computer. The results are summarised in Figure 3.

Computer	Running Time	# Samples	Estimate
1. Linux 2.0.35	60.04929 sec	512,283	0.393020
2. Irix 6.5	60.56188 sec	1,256,488	0.392972
3. Irix 5.3	60.76236 sec	153,642	0.393143
4. Irix 6.2	60.42300 sec	429,747	0.393073
5. Linux 2.0.27	60.91139 sec	145,177	0.393006
MASTER	60.91375 sec	2,497,337	0.39301172

Figure 3. Results of the `james` simulation.

Once again, the parallel algorithm proceeded efficiently and gave linear speed-up (aside from needing to discard the first 140 samples from each computer). Furthermore, because each computer allowed for an appropriate theoretical burn-in period, there was very little initialisation or burn-in bias (cf. Observation 10).

As an aside, we note that the resulting estimates depend heavily on the prior value b chosen. For example, if $b = 0$ as for a flat prior, then the resulting estimate of θ_1 is closer to 0.308.

6. Conclusion.

In this paper, we have argued that many typical Monte Carlo calculations, including i.i.d. Monte Carlo and also Markov chain Monte Carlo, can be run in parallel without great difficulty.

In a multiple-computer environment it is sometimes necessary to surrender some control, and to deal with computers and networks which may be unreliable, or down, or heavily used by other users. This is not an insurmountable obstacle, and we have argued that it is possible to handle these difficulties, at least to some extent, through analysis of the parallel computer set-up.

We believe that, as computers become more numerous, more easily available, and better networked, parallel Monte Carlo computations will become more and more common and more and more useful. We look forward to the exciting developments ahead.

Acknowledgements. I am very grateful to Alan J Rosenthal and to Radford Neal for discussing these issues with me in detail. I thank Peter Glynn, Neal Madras, Duncan Murdoch, Gareth Roberts, and Stu Whittington for helpful comments.

REFERENCES

[1] Beowulf Project. `http://www.beowulf.org/`

[2] S.P. Brooks and G.O. Roberts (1996), Diagnosing Convergence of Markov Chain Monte Carlo Algorithms. Preprint.

[3] K.M. Chandy, J. Kiniry, A. Rifkin, D. Zimmerman (1998), A framework for structured distributed object computing. Parallel Computing **24**, 1901–1922.

[4] M.K. Cowles (1998), MCMC Sampler Convergence Rates for Hierarchical Normal Linear Models: A Simulation Approach. Technical Report, Dept. of Statistics and Actuarial Science, University of Iowa.

[5] M.K. Cowles and B.P. Carlin (1996), Markov Chain Monte Carlo Convergence Diagnostics: A Comparative Review. J. Amer. Stat. Assoc. **91**, 883–904.

[6] M.K. Cowles, G.O. Roberts, and J.S. Rosenthal (December 1997), "Possible biases induced by MCMC convergence diagnostics". Preprint.

[7] M.K. Cowles and J.S. Rosenthal (1996), A simulation approach to convergence rates for Markov chain Monte Carlo algorithms. Stat. and Comput. **8** (1998), 115–124.

[8] Cray T90 Home Page. `http://www.sgi.com/t90/`

[9] The d'Artagnan Cluster. `http://turing.sci.yorku.ca/courseware/PHYS1010/dartagnan.html`

[10] Deschall. `http://www.frii.com/~rcv/deschall.htm`

[11] L. Devroye (1986), Non-uniform random variate generation. Springer-Verlag, New York.

[12] Distributed.net. `http://distributed.net/des/`

[13] B. Efron and C. Morris (1975), Data analysis using Stein's estimator and its generalizations. J. Amer. Stat. Assoc., Vol. **70**, No. **350**, 311-319.

[14] J.A. Fill (1998), An interruptible algorithm for perfect sampling via Markov chains. *Annals of Applied Probability*, 8:131–162.

[15] J.A. Fill and D.B. Wilson (1997), A Note about Time-dependent Sampling. Unpublished note.

[16] T.J. Fountain (1994), Parallel Computing: Principles and Practice. Cambridge University Press.

[17] A.E. Gelfand and A.F.M. Smith (1990), Sampling based approaches to calculating marginal densities. J. Amer. Stat. Assoc. **85**, 398-409.

[18] A. Gelman and D.B. Rubin (1992), Inference from iterative simulation using multiple sequences. Stat. Sci., Vol. **7**, No. **4**, 457-472.

[19] C.J. Geyer (1991), Markov chain Monte Carlo maximum likelihood. In *Computing Science and Statistics: Proceedings of the 23rd Symposium on the Interface*, 156–163.

[20] C.J. Geyer (1992), Practical Markov chain Monte Carlo. Stat. Sci., Vol. **7**, No. **4**, 473-483.

[21] C.J. Geyer and E.A. Thompson (1995), Annealing Markov chain Monte Carlo with applications to ancestral inference. J. Amer. Statist. Assoc. **90**, 909–920.

[22] W.R. Gilks, S. Richardson, and D.J. Spiegelhalter, ed. (1996), Markov chain Monte Carlo in practice. Chapman and Hall, London.

[23] P.W. Glynn and P. Heidelberger (1990), Bias properties of budget constrained simulations. Operations Research **38**, 801–814.

[24] P.W. Glynn and P. Heidelberger (1991), Analysis of Initial Transient Deletion for Replicated Steady-State Simulations. Operations Research Lett. **10**, 437–443.

[25] P.W. Glynn and P. Heidelberger (1991), Analysis of Parallel, Replicated Simulations under a Completion Time Constraint. ACM Trans. on Modeling and Simulation **1**, 3–23.

[26] P.W. Glynn and P. Heidelberger (1992), Analysis of Initial Transient Deletion for Parallel Steady-State Simulations. SIAM J. Scientific Stat. Computing **13**, 904–922.

[27] P.W. Glynn and P. Heidelberger (1992), Experiments with Initial Transient Deletion for Parallel Replicated Steady-State Simulations. Management Science **38**, 400–418.

[28] J.M. Hammersley and D.C. Handscomb (1964), Monte Carlo methods. John Wiley, New York.

[29] E.L. Lafferty, M.C. Michaud, M.J. Prelle, and J.B. Goethert (1993), Parallel Computing: An Introduction. Noyes Data Corporation, Park Ridge, New Jersey.

[30] E.L. Leiss (1995), Parallel and Vector Computing: A Practical Introduction. McGraw-Hill, New York.

[31] N. Madras (1998), Umbrella sampling and simulated tempering. In *Numerical Methods for Polymeric Systems* (Springer, New York; S. Whittington, ed.), 19–32.

[32] E. Marinari and G. Parisi (1992), Simulated tempering: a new Monte Carlo scheme. Europhys. Lett. **19**, 451–458.

[33] P. Matthews (1993), A slowly mixing Markov chain with implications for Gibbs sampling. Stat. Prob. Lett. **17**, 231-236.

[34] S.P. Meyn and R.L. Tweedie (1994), Computable bounds for convergence rates of Markov chains. Ann. Appl. Prob. **4**, 981-1011.

[35] C. Morris (1983), Parametric empirical Bayes confidence intervals. Scientific Inference, Data Analysis, and Robustness, 25-50.

[36] R. Motwani and P. Raghavan (1995), Randomized Algorithms. Cambridge University Press. (Chapter 12.)

[37] D.J. Murdoch and J.S. Rosenthal (1998), Efficient use of exact samples. Preprint.

[38] National Center for Supercomputing Applications. `http://www.ncsa.uiuc.edu/ncsa.html`

[39] R.M. Neal (1996), Sampling from multimodal distributions using tempered transitions. Stat. and Comput. **6**, 353–366.

[40] E. Orlandini (1998), Monte Carlo study of polymer systems by multiple Markov chain method. In *Numerical Methods for Polymeric Systems* (Springer, New York; S. Whittington, ed.), 33–57.

[41] Parallel Simulation Bookmarks. `http://www.cpsc.ucalgary.ca/~gomes/HTML/sim.html`

[42] Project Appleseed. `http://exodus.physics.ucla.edu/appleseed/appleseed.html`

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

[44] B.D. Ripley (1987), Stochastic simulation. Wiley, New York.

[45] G.O. Roberts and J.S. Rosenthal (1998), Markov chain Monte Carlo: Some practical implications of theoretical results (with discussion). Canadian J. Stat. **26**, 5–31.

[46] J.S. Rosenthal (1995), Minorization conditions and convergence rates for Markov chain Monte Carlo. J. Amer. Stat. Assoc. **90**, 558-566.

[47] J.S. Rosenthal (1996), Analysis of the Gibbs sampler for a model related to James-Stein estimators. Stat. and Comput. **6**, 269-275.

[48] J.S. Rosenthal. Parallel computation software.

`http://probability.ca/jeff/comp/`

[49] M.J. Schervish and B.P. Carlin (1992), On the convergence of successive substitution sampling. J. Comp. Graph. Stat. **1**, 111–127.

[50] A. Sinclair (1992), Improved bounds for mixing rates of Markov chains and multicommodity flow. Combinatorics, Prob., Comput. **1**, 351–370.

[51] A.F.M. Smith and G.O. Roberts (1993), Bayesian computation via the Gibbs sampler and related Markov chain Monte Carlo methods (with discussion). J. Roy. Stat. Soc. Ser. B **55**, 3-24.

[52] M.C. Tesi, D.J. Janse van Rensburg, E. Orlandini, and S.G. Whittington (1996), Monte Carlo study of the interacting self-avoiding walk model in three dimensions. J. Stat. Phys. **82**, 155–181.

[53] L. Tierney (1994), Markov chains for exploring posterior distributions (with discussion). Ann. Stat. **22**, 1701-1762.

[54] D.B. Wilson. Annotated bibliography of perfectly random sampling with Markov chains. `http://dimacs.rutgers.edu/~dbwilson/exact.html/`