Parallel Metropolis-Hastings Algorithms

Boyan Bejanov
bbejanov@bankofcanada.ca

April 25, 2014

Abstract

Prefetching is a parallelization approach for the Metropolis-Hastings algorithm, which is appropriate when the target probability density function is computationally expensive and impractical to parallelize. This project is an investigation into several aspects of the full and incomplete prefetching algorithms. Complexity analysis shows that prefetching is inefficient because it uses exponential number of processors, nevertheless it does provide speedup compared to the sequential algorithm. Computational experiments demonstrate the performance in practice. A data structure for the prefetching tree is designed in a way that simplifies the implementation of both full and incomplete prefetching. An attempt is also made to prove the P-completeness of a general Markov chain problem.

1 Introduction

The Monte Carlo methods (MC) are a class of numerical integration techniques, where the integral is interpreted as the expected value of some function of some random variable. The expectation can be statistically estimated from a sample drawn from the same distribution as the underlying random variable and this estimate serves as the numerical approximation of the integral. Needless to say, the MC methods are also applicable to problems other than numerical integration, so long as they can be reduced to the evaluation of an expected value.

The successful application of MC relies on the availability of numerical techniques for simulation of random samples from a distribution, which is given by its probability density function (pdf). We will denote this *target* pdf by $\pi(x)$. It is always assumed that a stream of independent random numbers with uniform distribution in the interval $(0, 1)$ is available without ado. For some of the other common probability distributions there are direct methods of simulation, which perform some computation on the uniform random numbers, transforming them into random numbers of the desired distribution. However, these techniques are limited to special cases. One of the most important general sampling technique is the Acceptance-Rejection sampling (AR). Each iteration of the AR goes in two steps. We first simulate a random number, X, from some other distribution, which can be simulated by an already known method. We call this distribution *proposal* and denote its pdf by $q(x)$. In the second step, we simulate a uniform random number and compare it to the adjusted ratio of the target and proposal densities. Based on this comparison, we either accept X as a random number from π, or we reject and discard it. The AR method is embarrassingly parallel, since multiple random numbers can be simulated on multiple processors simultaneously. In addition, it produces a sample of independent random numbers, which can be

used directly in an MC method. However, its applicability is limited to the cases where $\pi(x)$ is given explicitly by a known formula. Even then, the computational efficiency of AR (as measured by the acceptance rate) depends on the choice of the proposal, $q(x)$, and a tight estimate of the adjustment constant in the AR test, both of which can require some non-trivial mathematical derivations, especially for multivariate distributions.

The method of choice in the cases of a multivariate distribution of high dimension, or when the formula for the target pdf is too complicated, or when the target pdf can only be computed numerically, is the Markov Chain Monte Carlo method (MCMC). A Markov Chain is a process, which is described by a *state space*, i.e. the set of all possible values the process can take, and a transition rule, which may depend only on the current state, but not any earlier ones. It is known from the theory of Markov chains that, if certain conditions are satisfied, the process has an *invariant distribution* and the distributions of successive states converge to the invariant distribution. In the MCMC method, we construct a transition rule, such that the resulting Markov chain has an invariant distribution, and this invariant distribution is exactly $\pi(x)$. The initial part of the chain, before the distribution of states gets close enough to the invariant distribution, is known as *burn-in* and is discarded. The remaining chain can be used as the random sample in a Monte Carlo method, although with the caveat that it is not an independent random sample, so the auto-correlation of the process must be taken into account.

The AR, MCMC and other methods for simulation of random variables, as well as the underlying mathematical theories, can be found in [12].

The Metropolis-Hastings algorithm (M-H) combines AR and MCMC. Effectively, it is a recipe for constructing the transition rule of a Markov chain given its desired invariant distribution. Once again, we have a proposal distribution, however this time the proposal distribution depends on the current state of the chain, i.e. $q = q(x; y)$. The good news is that the accept-reject test is a ratio of the values of q and π at the current and the proposed points with no adjustment constants to be derived in advance. If the proposed point is accepted, then it becomes the next state of the chain, otherwise the next state is equal to the current state, i.e. no points are discarded as it was the case in the AR sampling method. An intuitive and self-contained presentation of the Metropolis-Hastings method can be found in [4].

Other than Monte Carlo type estimations of expected values, the M-H is a general method for simulating random samples from distributions that are otherwise impossible to simulate. It is particularly popular for Bayesian analysis of complex stochastic models. Such models always involve a number of parameters, which rarely can be derived from first principles. More frequently, the parameters are estimated from observed data using maximum-likelihood, or some other methods of statistical estimation. When the model is sufficiently complex, the usual inference based on confidence intervals for the estimated parameters is impossible to derive. In such cases, the approach of Bayesian statistics is to consider the parameters as random variables, simulate a sample from their joint distribution, and use this sample to infer the properties of the distribution that are of interest.

The specific focus in this project is the application of M-H to Bayesian inference of large Dynamic Stochastic General Equilibrium models (DSGE), which are used in the field of macro-economics ([15]). The likelihood function is computed using a Kalman filter, which can be computationally very expensive, especially in the nonlinear case. Therefore, when considering various methods, we will keep in mind the (simplifying) assumption that the evaluation of $\pi(x)$ is much more time consuming than all the other computations in a single step of the Markov chain (such as the computation of $q(x, y)$, simulation from $q(x, y)$,

individual arithmetic operations).

The objectives of this project include

- Survey the existing methods and evaluate them.
- Identify the most promising method, for the particular setting we have, and implement it.
- Investigate the benefits of parallelization in terms of theoretical and practically achieved speedup.
- Consider the question of P-completeness of the M-H algorithm.

2 Literature Review

The search for parallel algorithms for M-H and other MCMC techniques has received much attention from the computational statistics community as well as from researchers in various fields where MCMC finds real-world applications. Unfortunately, there seems to be little attention given to this question in the world of computer science.

In [13] the authors propose an *adaptive* M-H method for estimation of a climate model, where multiple Markov chains are simulated in parallel. In an adaptive MCMC algorithm, the computed values of the target pdf are used to construct a better proposal distribution. Each chain must undergo its own burn-in, therefore the time before the algorithm starts yielding useful output is the same as in the sequential case. This greatly limits the gain in speedup due to parallelization. However, the periodic updates to the proposal distribution in each chain use the values of the target density produced by all chains. This way the algorithm produces higher quality proposal compared to the sequential version, thus improving the *statistical efficiency* of the chain.

Another recent application of parallel MCMC is presented in [17] in the context of Bayesian models of animal breeding and genetics. Here the authors also consider multiple Markov chains running in parallel and acknowledge that this is only applicable to single-parameter models and simpler models with few parameters, where the burn-in is small. In the case of more complex models, they use parallel computation of $\pi(x)$ by leveraging specific properties of the particular model they consider.

A modified M-H algorithm is presented in [9]. It simulates a single Markov chain with improved statistical properties by considering multiple proposal points at each step. The proposals are computed in parallel and one of them is randomly selected to be the next state. This allows for better coverage of the state space, i.e. reduced burn-in time. However, the time for computing one step is the same as in the sequential M-H algorithm.

In [16], the authors propose a parallel MCMC method which is only applicable to numerical integration. The state space is partitioned into disjoint regions and the total integral is evaluated as the sum of the integrals over the individual regions. A separate Markov chain is simulated for each region and all chains run in parallel. The clever innovation of this method is that it allows the chains to leave their respective regions and are recombined afterwards. This considerably simplifies previous methods using partitioning, which must contain the individual chains within their regions. Unfortunately, this method is not applicable in our case, since it only computes the expectation, but doesn't produce a random sample.

In general the parallel MCMC methods can be divided into methods using multiple independent chains that are simulated in parallel and methods that simulate a single chain in parallel. All of the methods discussed so far are in the first category. The paper [7] contains a recent and detailed overview of parallel MCMC methods based on multiple chains.

The single chain parallel methods can be further subdivided into *within draw* and *between draw* parallelization, as suggested in [14]. In the former class of methods multiple processors collaborate on a single computation of the target density. Clearly this approach is problem-specific. The between draw parallel methods are characterized by considering multiple future steps of the chain and pre-computing the necessary values of the target density in parallel.

A Metropolis-Hastings algorithm whose proposal density does not depend on the current state of the Markov chain is called *Independence* Metropolis-Hastings (IMH). In this case the between-draw parallelization is straight-forward. Unfortunately, a good independence proposal is difficult to construct beforehand. Instead, in the method proposed in [15], an independence proposal is estimated from the values of π that have already been computed. As the chain runs, the independence proposal is adapted to any new information about the target density. However, the Markov chain must start with a non-independence proposal. At each adaptation point, the adapted proposal is constructed as a mixture of the newly re-estimated independence proposal and the non-independence proposal, with the relative weight of the independence proposal gradually increasing. See also [8] and [6], which suggest similar algorithms with different approaches to constructing the independence proposal. Many other examples of adaptive MCMC can be found in [11].

A regeneration time in a Markov chain is a time when the state is independent of the history of the chain up to that point. The pieces of the chain between two regeneration times are called *tours* and are independent and identically distributed entities. This allows multiple tours of the chain to be simulated in parallel, each starting from a regeneration time and running until the next regeneration time is encountered. The tours are then concatenated to form a single long chain. Although a wide class of Markov chain are regenerative, the identification of regeneration points is very difficult (cf. [10] and [5]). More recently, [2] proposed a constructive method of identifying regeneration times in the context of parallel M-H. Using regeneration is only advantageous for low- and moderately-dimensional distribution, because the expected length of the tour increases dramatically with dimension.

A recent algorithm, proposed in [1], is called prefetching. Suppose the Markov chain of an M-H algorithm is at some state X_n at some time n. All possible trajectories for the next h steps form a full binary tree with root X_n and depth h. At each node the two edges correspond to "accept" and "reject". There are 2^h possible paths and a total of 2^h different proposal states in the tree. The idea of prefetching is to compute the values of π at the 2^h proposals in parallel and then perform the h accept-reject steps sequentially on the master process. In [14] the authors suggest an improvement of this algorithm, whereby only the most probable paths are prefetched, thus reducing the number of required processors, although at the risk that fewer than h steps may be completed. Another method suggested in [3] can be interpreted as a special case of the methodology suggested in [14]. The authors leverage the fact that the optimal acceptance rate (from probabilistic point of view) is usually low and prefetch only the "reject" branches.

3 Project Report

3.1 P-Completeness

It is interesting to consider the question of whether the simulation of a Markov chain is a P-complete problem. Surprisingly, this question doesn't seem to have been addressed – literature search yields no hits.

Claim 1 *The simulation of a Markov chain is a P-complete problem*

Proof. Let G be an instance of the generability problem: "given a set X with $|X| = n$, $T \subseteq X$, $x \in X$ and a binary operation $*$, is x in the closure of T under $*$?" We will construct a Markov chain that solves this generability problem.

We start by introducing an arbitrary labeling of the different elements of X with different integers from 0 to $n-1$. In fact, without loss of generality, we will assume in what follows that $X = \{0, \ldots, n-1\}$.

The states of our Markov chain will be triples (A, k, c), where A is a subset of X, k is an element of X, and c is a true/false flag. The set A is the portion of the closure of T computed so far. The element k will allow us to keep track of which element we need to consider next. The flag c will keep track of whether the set A has changed since the last time we visited x.

Suppose the current state is (A, k, c). The transition rule for the Markov chain is deterministic and follows these guidelines.

- The next A will be either A itself or A with k added to it. We add k only if it isn't already in A and can be generated from it, i.e. there are $i, j \in A$ s.t. $i * j = k$.
- The next k will always be the next element of X in a circular order, i.e. $k+1 \bmod n$.
- If k was added to A then the next state will have $c = true$, otherwise we don't change c.

The initial state will be $(T, x, true)$.
In order to guarantee the proper termination of the algorithm we add the following rules.

- If $k = x$ and $c = true$ then check if x is in the closure of A. If so then stop, otherwise transition to $(A, x+1 \bmod n, false)$.
- If $k = x$ and $c = false$ then stop.

If the chain stops because of the first bullet, then x is in the closure of T. If the chain stops because of the second bullet, then A is exactly the closure of T, which doesn't contain x.

Notice that this is a proper Markov chain because the transition rule uses only the current state and not any of the previous history. Also, the computationally expensive part of this transition rule is the computation of the problem of whether k is in the closure of A. This can be solved in constant time on $O(n^2)$ processors, i.e. the operation is NC.

□

3.2 Notation

In what follows, we denote $\pi(x)$ the probability distribution function of the target probability density, where $x \in \mathbf{R}^d$ is a d-dimensional vector of real numbers. The proposal probability kernel will be denoted $q(x, y)$. The first argument of q stands for the current state. For a given and fixed x, the kernel $q(x, y)$, as a function of y alone, reduces to a probability density function. By abuse of notation, we will denote this probability distribution by $q(x, \cdot)$.

3.3 The sequential Metropolis-Hastings algorithm

The Metropolis-Hastings algorithm provides a transition rule such that the resulting Markov chain will have stationary distribution $\pi(x)$.

Algorithm 1 Metropolis-Hastings
Inputs: target pdf $\pi(x)$, proposal kernel $q(x, y)$ and its proposal distribution $q(x, \cdot)$, initial state S_0, length of desired chain n.

0. Compute $\pi(S_0)$.
1. For k from 0 to $n - 1$
2. Generate candidate $Y \sim q(S_k, \cdot)$.
3. Compute $\alpha = \dfrac{\pi(Y)q(Y, S_k)}{\pi(S_k)q(S_k, Y)}$.
4. Generate $U \sim \text{Uniform}[0, 1]$
5. If $U < \alpha$ then *accept* and set $S_{k+1} = Y$
   else *reject* and set $S_{k+1} = S_k$.
6. Next k

Output: Markov chain of length n: $S_0, S_1, \ldots, S_n$.
— End of Algorithm —

If $q(x, y)$ were a transition kernel that would create a Markov chain with stationary distribution π, then the numerator of α would be exactly the unconditional probability that the chain would transition from state Y to state S_k. Similarly the denominator would be the unconditional probability of the reverse jump occurring. These two probabilities being equal is a sufficient condition for $q(x, y)$ to be a transition kernel that would run the Markov chain into stationary distribution π.

If the value of α computed in step 3 is less than 1, this means that the denominator is larger than the numerator, i.e. $q(x, y)$ is generating more transitions from S_k to Y than from Y to S_k. The M-H algorithm "corrects" $q(x, y)$ by rejecting a fraction α of such jumps, thus making the probabilities of the two jumps equal.

Note that the computation of α in step 3 requires only a single evaluation of the target density at the candidate point, $\pi(Y)$. The value of $\pi(S_k)$ is available from the previous iteration.

3.4 Prefetching algorithm

In the original prefetching algorithm proposed in [1] we consider all possible paths that the Markov chain may take h steps into the future. At each step a new candidate is proposed and is either accepted or rejected. It is clear that all possible paths form a complete binary tree with depth h. This is illustrated in figure 1.

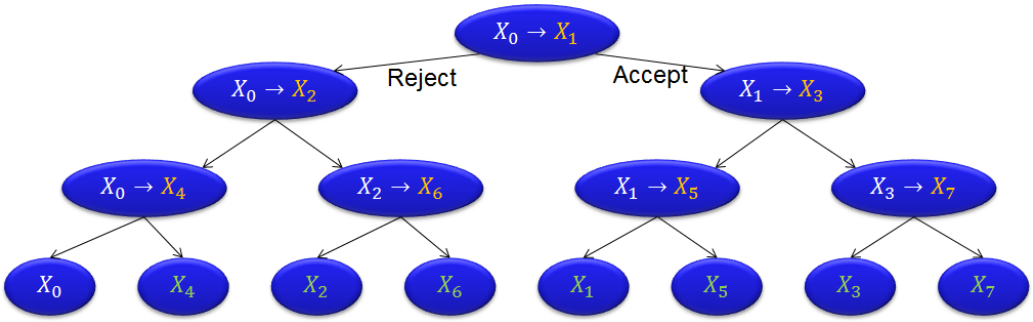

Figure 1: Prefetching tree includes all possible paths the chain may take h steps into the future. White shows current state. Orange shows proposed candidate.

Suppose that the current state is S_k. This will be the root of the prefetching tree, so we set $X_0 = S_k$. We see that all possible states the the chain may visit in the h steps are 2^h in total. Of these, one state is X_0 for which the value $\pi(X_0) = \pi(S_k)$ is already known. The original prefetching algorithm is to compute the values of π at the remaining $2^h - 1$ points in parallel on $2^h - 1$ processors and then perform h steps of the sequential M-H algorithm. The chain will end up in one of the states at the bottom of the tree which will be state S_{k+h} of the chain. We build a new prefetching tree with root S_{k+h} and depth h and so on.

3.5 Complexity analysis of prefetching

It was already mentioned that we are working under the assumption that the computation of $\pi(x)$ takes more time than any other calculation in the algorithm. In particular, the generation of random variables from the proposal distribution $q(x, \cdot)$, the evaluation of the proposal kernel $q(x, y)$, the generation of uniform random variables and all other operations take negligibly small amount of time compared to $\pi(x)$. Under this assumption, the time of the algorithm is adequately modelled by the number of evaluations of $\pi(x)$.

The problem size is n, which denotes the number of steps of the Markov chain to be computed. The depth of the prefetching tree is h and we assume that n is an integral multiple of h.

The time of the sequential M-H algorithm is n, since it requires one evaluation of $\pi(Y)$ at each step. This ignores the evaluation of $\pi(S_0)$ – we may assume that this value is provided as input to the algorithm. This simplifies the equations a bit and does not change the conclusions. Sequential time is

$$T_s = n. \tag{1}$$

For the parallel time, we first consider the case where the processors equal the number of required evaluations of π, i.e.

$$p = 2^h - 1 \tag{2}$$

Then the evaluation of a single prefetching tree takes time 1 and we have n/h trees in total, so the parallel time is

$$T(p) = \frac{n}{h} \tag{3}$$

This yields the following speedup and efficiency:

$$s(p) = \frac{T_s}{T(p)} = h, \qquad e(p) = \frac{s(p)}{p} = \frac{h}{2^h - 1}. \tag{4}$$

We see that the speedup is h and if we solve the relationship (2) for p we may conclude that the speedup is logarithmic in p, as has been suggested in [1] and [14]. This, in fact, is not correct and we can see this in the case where p is not linked to h.

Now consider the general case where p is not related to h. The parallel time now depends on both p and h and, therefore, so do the speedup and efficiency. The work to be done for a single prefetching tree is still $2^h - 1$, but this time the time is $(2^h - 1)/p$ on p processors by simulation. So the total parallel time for n/h prefetching trees is

$$T(h, p) = \frac{n(2^h - 1)}{hp}. \tag{5}$$

The resulting speedup and efficiency are

$$s(h, p) = \frac{hp}{2^h - 1} \qquad \text{and} \qquad e(h, p) = \frac{h}{2^h - 1}. \tag{6}$$

We see now that the speedup is actually linear in p. However, the slope of the line, which equals the efficiency, is small and decreases exponentially as h increases. By this consideration, the smallest value of h will give the most efficient version of prefetching, which ironically is $h = 1$, i.e. the sequential M-H.

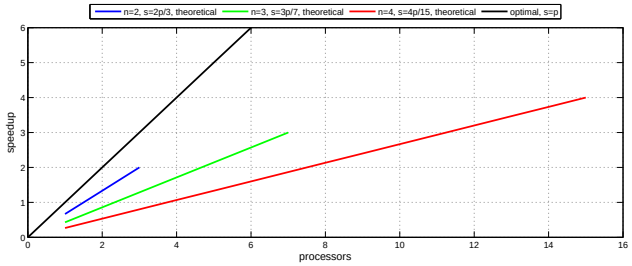

Figure 2: Theoretical speedup curves for prefetching

3.6 Implementation

There is only one data structure in the prefetching algorithm – the prefetching tree. We perform three operations on it, which are listed below.

build This includes generating the correct candidates for each node.

prefetch This is the evaluation of the values of π.

travel This is the operation of running h steps of M-H using the prefetched proposals and their π values.

These operations can be simplified by proper numbering of the X-points within the nodes in the prefetching tree. With depth h, there are 2^h different X's and their indexes are the h-bit integers. We adopt the strategy that at step s we set bit in position $(s-1)$ to either 1 for the accept branch, or to 0 for the reject branch. The bits positions are numbered from right to left starting from 0. At step s, all bits to the right of bit $(s-1)$ are already set to some pattern of 0s and 1s corresponding to the rejections and acceptances along the

path from the root to the node. All bits to the left of bit $(s-1)$ have not been touched yet and are still 0. The proposed candidate has index derived from the index of the current state by setting bit $(s-1)$ to 1. This is also illustrated in figure 1.

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
s	/	1	2		3				4							
c		0	0	1	0	1	2	3	0	1	2	3	4	5	6	7

Figure 3: Relationships between the step number s, the current state c and the proposed state k associated with a node in the prefetching tree (cf. figure 1).

When building the tree, for each point, say X_k, we need to know at which step it was proposed and what was the current state, denote it X_c, when that happened. Notice that all candidates proposed at step s have 1 in bit position $(s-1)$ and all bits further to the left are 0. This means that the step are which a point was proposed is the position of the most significant bit that is set to 1. The point that was the current state has index in which the bit at position $(s-1)$ is set to 0 and all the other bits are the same. These considerations are summed up in the following formulas, and are also illustrated in figure 3.

$$s(k) = \begin{cases} 0, & k = 0 \\ 1 + \lfloor \log k \rfloor, & k > 0 \end{cases}$$
$$c(k) = k - 2^{s(k)-1}, \qquad \text{for} k > 0$$

Here $s(k)$ denotes the step number and $c(k)$ denotes the index of the point that was the current state at the time when point with index k was proposed.

The following code **builds** the tree, i.e. generates proposals and places them in an array.

```
for(c=0; c< (1<<(h-1)) ; +=c) {
    for(s=s(c); s<h; ++s) {
        k = c + (1<<s);
        generate X[k] ~ q(X[c],*)
    }
}
```

The **prefetching** of the tree is straight forward; it simply involves the evaluation of π at $2^h - 1$ points in parallel. The following code example uses Cilk.

```
cilk void prefetch_target_values(int c, int s) {
    if(s==h) {
        if (c>0) compute pi(X[c]);
    } else {
        spawn prefetch_target_values(c, s+1);
        spawn prefetch_target_values(c+(1<<s), s+1);
    }
    cilk_sync;
}
```

To **travel** the tree we need the relationship between the current node and the proposed candidate at the current step. The only difference between the indexes of these two points is the bit in position $s-1$.

```
c = 0;
for(s=1; s<h; ++s) {
    p = c + (1<<(s-1));
    if (accept X[p]) c = p;
    S[s] = X[c];
}
```

3.7 Run times

The algorithm was tested on a synthetic problem where the target distribution was chosen to be a 15-dimensional mixture of two normal distributions. Because the pdf of this distribution is relatively simple and quick to evaluate, additional computations were added to π to increase its computation time to approximately 0.1 sec. The test server has two Xeon E5-2690 CPUs with 8 cores each running at 2.90GHz for a total of 16 cores.

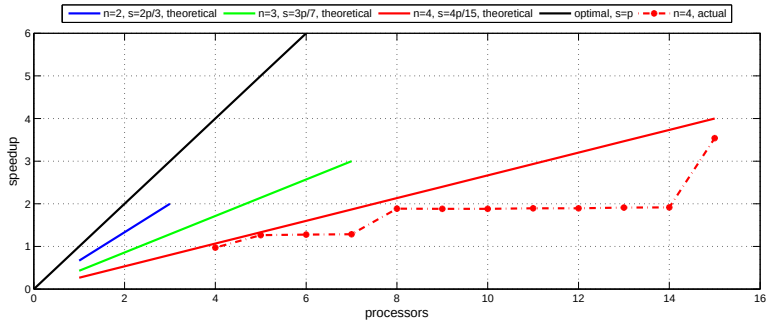

Figure 4: The empirical speedup curve

In figure 4 we see the observed speedup curve for $h = 4$. It is actually a step function that approximates from below the straight line of the theoretical speedup. The step function is the result of not applying the simulation technique in order to balance the load evenly among the cores. For example, when the number of processors is between 8 and 14, some of the processors compute two values of π while the other processors compute just one value of π after which they sit idle waiting for the processors with the higher loads to finish. Because of this, the speed up for all p between 8 and 14 processors is 2.

3.8 Incomplete prefetching

The incomplete prefetching was suggested in [14]. It aims to improve the efficiency of the method by computing only a part of the prefetching tree. The advantage is that the number of processors used can be less than exponential. On the other hand, there is no guarantee that the chain will travel h steps through the tree, because there is a positive probability that the chain may encounter a node that has not been prefetched.

By changing the variance of the proposal kernel $q(x, y)$ we are able to target a particular acceptance rate. The target acceptance rate is usually denoted α, although it should be

distinguished from the α that appears in step 3 of the M-H algorithm. The latter is a part of the algorithm and necessarily is different at each step since the current state and the proposed candidate change. The former α is the actual average number of accepted proposals over the long run of the chain under the M-H algorithm. Usually the acceptance rate is targeted to be some value that is optimal from the point of view of the statistical quality of the Markov chain. For multidimensional distributions with dimension higher than 6, $\alpha = 0.234$ is optimal in vast majority of the cases.

The known value of the targeted, and therefore expected, acceptance rate allows us to assign probabilities to the accept and reject branches of the prefetching tree. All accept branches have probability α and the reject branches have probability $(1 - \alpha)$.

The probability that a node will actually be visited by the chain is the product of the probabilities assigned to the edges along the path from the root to node. If we have p processors, we evaluate π at p proposals, which we choose to maximize the expected number of steps (or depth) that the chain will travel through the computed part of the tree.

When the chain visits a node of the tree, it will need the value of π at the point that is proposed at that node (the X_k marked in orange in figure 1). So selecting a node to be prefetched means that $\pi(X_k)$ is evaluated. Note that a node can only be reached if all nodes connecting it to the root are also being prefetched.

The expected number of steps for a particular selection of points to be prefetched is simply the sum of the probabilities of all selected nodes. This fact is not difficult to prove, however the proof is omitted here. The highest expected depth for given number of points, p, will be denoted $D(p)$.

All claims about the node probabilities and the expected depth made in this section are justified in [14].

3.9 Implementation remarks for incomplete prefetching

For incomplete prefetching we need two additional operations on the prefetching tree.

probability assignments This is the task of computing the probability for a node to be visited by the chain.

selection of nodes This is the operation of selecting the p nodes most likely to be visited.

The root of the tree is always visited, so its probability is 1.0. The proposal there is X_1, so its probability is also 1.0. Following the accept branch, we assign probability α to X_3, while following the reject branch we assign probability $(1 - \alpha)$ to X_2. Now from X_3, which already has probability α, if we follow the accept branch, we multiply by α and assign probability α^2 to X_7. Similarly, following the reject branch we assign probability $\alpha(1 - \alpha)$ to X_5. We continue this pattern throughout the tree.

What is the formula that for a given X_k (a proposal, orange in figure 1) will compute the indexes of the accept and reject branches from X_k? We know the step where X_k is proposed: it is the same $s(k)$ as before. The accept branch will lead us to the point we propose at step $(s(k) + 1)$ when X_k will have become the current state. The index is k with the $s(k)$-th bit set to 1, i.e. the accept branch of k leads to $a(k)$ given by

$$a(k) = k + 2^{s(k)}. \tag{7}$$

The reject branch is also a proposal at step $s(k) + 1$, so it will also have the $s(k)$-th bit set to 1. However, its $(s(k) - 1)$-st bit will be zero, since this is the case where X_k would have

been rejected at step $s(k)$. So the formula for the reject branch is

$$r(k) = a(k) - 2^{s(k)-1}. \tag{8}$$

This leads to the following **probability assignment** code.

```
    prob[1] = 1.0;
    for(k=1; k < (1<<(h-1)); ++k) {
        s = log2(k)+1;
        a = c + (1<<s);
        r = a - (1<<(s-1));
        prob[a] = prob[k] * alpha;
        prob[r] = prob[k] * (1.0-alpha);
    }
```

In the selection of most probable nodes to be visited, we need a connected sub-tree with p nodes. The naive approach is to start with an empty sub-tree. In each step, starting from the root, we visit nodes that are one edge away from the sub-tree we have so far. The node with the highest probability gets added to the sub-tree and the process repeats until we have p nodes selected. The following code is an example of the **selection of nodes** operation.

```
int best_point(int c) {
    int s = log2(c)+1;
    if (s > h) return 0;
    if ( !selected[c] ) return c;
    int best_accept = best_point[ c + (1<<s) ];
    int best_reject = best_point[ c + (1<<s) - (1<<(s-1)) ];
    if (prob[best_accept] >= prob[best_reject])
        return best accept;
    else
        return best_reject;
}

void select_points() {
    for(int i=0; i<(1<<h); ++i) select[i]=0;  // deselect all
    select[0] = 1;  // sentinel, point 0 is never evaluated
    prob[0] = 0.0;  // sentinel
    for(int i=0; i<p; ++i) {
        int k = best_point(1);
        select[k] = 1;
    }
}
```

3.10 Complexity of the incomplete prefetching

The time to compute one incomplete prefetching tree is 1, since we always select p points to prefetch. With each incomplete tree we expect to make $D(p)$ steps, therefore we expect

to have to compute $n/D(p)$ trees. Therefore the expected parallel time is

$$T(p) = \frac{n}{D(p)}. \tag{9}$$

From here, the speedup is

$$s(p) = \frac{T_s}{T(p)} = \frac{nD(p)}{n} = D(p). \tag{10}$$

In figure 5 the solid line shows the speedup curve for $\alpha = 0.234$. If we compare it to the curves in figure 2, we see that the expected speedup of incomplete prefetching dominates that of full prefetching regardless of the value of h.

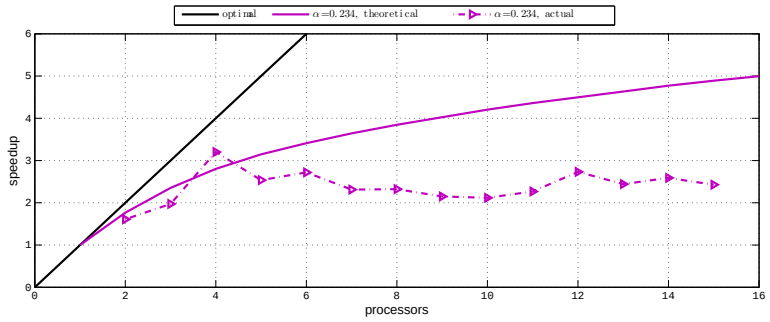

Figure 5: Theoretical and empirical speedup curves for incomplete prefetching with $\alpha = 0.234$.

3.11 Run times for incomplete prefetching

The dashed line in figure 5 shows the empirical results from running the incomplete prefetching algorithm on the same example as in the full prefetching case. The curve flattens off and does not follow very closely the theoretical one. This may be attributed to the fact that the theoretical curve is not a guaranteed performance, rather it is the expected speedup. This expected speedup is calculated assuming certain acceptance rate in the M-H algorithm. If the realized acceptance rate is not close to this assumed value, then the actual performance would reflect that. Nevertheless, the graphs suggest that with incomplete prefetching we can achieve speedup of over 3 using 5 processors, while speedup of over 3 in the full prefetching is achieved with $h = 4$ and 15 processors.

4 Conclusion

Prefetching is a promising new algorithm with somewhat limited applicability. It only makes sense to use prefetching when the computation of the target probability is very expensive and it is not possible to parallelize π itself. Even then, although there is speedup to be gained from prefetching, it comes at the cost of having to use exponential number of processors. This makes prefetching extremely inefficient; in fact the efficiency diminishes exponentially as h increases.

Incomplete prefetching attempts to improve the efficiency by computing only a part of the prefetching tree. The run times show that this approach is able to achieve the same or

slightly better run times and speedups, but using far fewer processors compared to the full prefetching. At the same time, the performance in practice does not achieve the expected values, although this may be possible to improve by more careful selection of the proposal kernel.

References

[1] AE Brockwell. Parallel Markov Chain Monte Carlo simulation by pre-fetching. *Journal of Computational and Graphical Statistics*, 15(1):246–261, 2006.

[2] Anthony E Brockwell and Joseph B Kadane. Identification of regeneration times in MCMC simulation, with application to adaptive schemes. *Journal of Computational and Graphical Statistics*, 14(2), 2005.

[3] Jonathan MR Byrd, Stephen A Jarvis, and Abhir H Bhalerao. Reducing the run-time of MCMC programs by multithreading on SMP architectures. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

[4] Siddhartha Chib and Edward Greenberg. Understanding the Metropolis–Hastings algorithm. *The American Statistician*, 49(4):327–335, 1995.

[5] Walter R Gilks, Gareth O Roberts, and Sujit K Sahu. Adaptive Markov Chain Monte Carlo through regeneration. *Journal of the American Statistical Association*, 93(443):1045–1054, 1998.

[6] Paolo Giordani and Robert Kohn. Adaptive independent Metropolis–Hastings by fast estimation of mixtures of normals. *Journal of Computational and Graphical Statistics*, 19(2):243–259, 2010.

[7] Guangbao Guo. Parallel statistical computing for statistical inference. *Journal of Statistical Theory and Practice*, 6(3):536–565, 2012.

[8] Lars Holden, Ragnar Hauge, and Marit Holden. Adaptive independent Metropolis–Hastings. *The Annals of Applied Probability*, pages 395–413, 2009.

[9] Guthrie Miller. Markov Chain Monte Carlo calculations allowing parallel processing using a variant of the metropolis algorithm. *Open Numer Methods J*, 2:12–7, 2010.

[10] Per Mykland, Luke Tierney, and Bin Yu. Regeneration in Markov chain samplers. *Journal of the American Statistical Association*, 90(429):233–241, 1995.

[11] Gareth O Roberts and Jeffrey S Rosenthal. Examples of adaptive MCMC. *Journal of Computational and Graphical Statistics*, 18(2):349–367, 2009.

[12] Sheldon M. Ross. *Simulation, Fourth Edition*. Academic Press, Inc., Orlando, FL, USA, 2006.

[13] Antti Solonen, Pirkka Ollinaho, Marko Laine, Heikki Haario, Johanna Tamminen, and Heikki Järvinen. Efficient MCMC for climate model parameter estimation: Parallel adaptive chains and early rejection. *Bayesian Analysis*, 7(3):715–736, 2012.

[14] Ingvar Strid. Efficient parallelisation of Metropolis–Hastings algorithms using a prefetching approach. *Computational Statistics & Data Analysis*, 54(11):2814–2835, 2010.

[15] Ingvar Strid, Paolo Giordani, and Robert Kohn. Adaptive hybrid Metropolis-Hastings samplers for DSGE models. Technical report, SSE/EFI Working Paper Series in Economics and Finance, 2010.

[16] Douglas N VanDerwerken and Scott C Schmidler. Parallel Markov Chain Monte Carlo. *arXiv preprint arXiv:1312.7479*, 2013.

[17] Xiao-Lin Wu, Chuanyu Sun, Timothy M Beissinger, Guilherme JM Rosa, Kent A Weigel, Natalia de Leon Gatti, and Daniel Gianola. Parallel Markov chain Monte Carlo – bridging the gap to high-performance Bayesian computation in animal breeding and genetics. *Genetics Selection Evolution*, 44(1):29, 2012.