On the Parallelisation of MCMC by Speculative Chain Execution

Jonathan M. R. Byrd
Department of Computer Science
University of Warwick
Coventry, CV4 7AL, UK
Email: J.M.R.Byrd@dcs.warwick.ac.uk

Stephen A. Jarvis
Department of Computer Science
University of Warwick
Coventry, CV4 7AL, UK
Email: Stephen.Jarvis@dcs.warwick.ac.uk

Abhir H. Bhalerao
Department of Computer Science
University of Warwick
Coventry, CV4 7AL, UK
Email: Abhir.Bhalerao@dcs.warwick.ac.uk

***Abstract*—The increasing availability of multi-core and multi-processor architectures provides new opportunities for improving the performance of many computer simulations. Markov Chain Monte Carlo (MCMC) simulations are widely used for approximate counting problems, Bayesian inference and as a means for estimating very high-dimensional integrals. As such MCMC has had a wide variety of applications in fields including computational biology and physics, financial econometrics, machine learning and image processing.**

One method for improving the performance of Markov Chain Monte Carlo simulations is to use SMP machines to perform 'speculative moves', reducing the runtime whilst producing statistically identical results to conventional sequential implementations. In this paper we examine the circumstances under which the original speculative moves method performs poorly, and consider how some of the situations can be addressed by refining the implementation. We extend the technique to perform Markov Chains speculatively, expanding the range of algorithms that maybe be accelerated by speculative execution to those with non-uniform move processing times. By simulating program runs we can predict the theoretical reduction in runtime that may be achieved by this technique. We compare how efficiently different architectures perform in using this method, and present experiments that demonstrate a runtime reduction of up to 35-42% where using conventional speculative moves would result in execution as slow, if not slower, than sequential processing.

I. Introduction

Markov Chain Monte Carlo (MCMC) is a computationally intensive iterative technique for sampling from a (typically very large) probability distribution. It is commonly applied to calculating multi-dimensional integrals and for conducting Bayesian inference, allowing prior knowledge to guide the analysis of input data. As such it has have numerous applications in Bayesian statistics, computational physics and computational biology. Notable examples include constructing phylogenetic trees [1], spectral modeling of X-ray data from the Chandra X-ray satellite [2], and for calculating financial econometrics [3].

MCMC using Bayesian inference is particularly suited to problems where there is prior knowledge of certain aspects of the solution. For instance, when counting tree crowns in satellite images where the trees will mostly be be arranged in a regular pattern [4]. By incorporating expected properties of the solution, the stability of the simulation is improved and the chances of consistent false-positives is reduced. MCMC is also good at identifying similar but distinct solutions (i.e. is an artifact in a blood sample one blood cell or two overlapping cells) and giving the relative probabilities of these different interpretations of the original data.

The main limitation of the MCMC method is its runtime, particularly when processing large images. The time per iteration can increase exponentially with the number artifacts to be found in the image, and the total number of iterations required increases with both the number of artifacts and the size of the image. As an example, the mapping of vascular trees in retinal images as detailed in [5], [6] took upwards of 4 hours to converge when run on a 2.8GHz Pentium 4, and takes much longer to explore alternative modes (additional potential interpretations for the input data). The practicality of such solutions (in real-time clinical diagnostics for example) is therefore limited. High throughput microscopy applications face a similar problem, although individual images may be processed quickly, the large number of images to analyse make reductions in runtime desirable

The MCMC method is inherently sequential, each iteration must depend only on its predecessor to conserve the statistical properties that guarantee eventual convergence on the desired solution. Because of this, most conventional parallelisation methods cannot effectively be applied. One parallelisation method that does work is to make use of multicore or multiprocessor hardware to perform moves (iterations) speculatively, as presented in [7]. Speculative moves is a general purpose method of implementing MCMC that is invisible to the statistical algorithm being employed. Using readily available quad-core machines the runtime of a standard MCMC program can be reduced by up to 63%, or a reduction of 43% if only a dual-core machine is available. There are however situations where speculative moves are ineffectual, and it is these cases that are the focus of this paper.

The contributions of this paper are fourfold:

- We calculate the consequence of a "naive" implementation of speculative moves where different expected execution times exist for different types of move, and find this results in a suboptimal reduction in runtime (in some cases, a net increase in runtime may result).
- We present a method of implementing speculative moves that partially remove this excessive runtime caused by the

varying move execution times.
- We present a new method that speculatively considers entire Markov Chains to further improve the runtime of applications with varying move execution times.
- We demonstrate how the runtime of MCMC applications using 'speculative chains' can be predicted by use of a simulator program, and compare how efficiently a number of computer architectures were able to apply speculative chains to a practical application.

The remainder of this paper is organised as follows. In section II we explain the MCMC method and the difficulties in parallelising it. Section III reviews the current forms of parallel MCMC, with special attention to the speculative moves method. Our criticisms and shortcomings of the speculative moves method are covered in section IV, and our implementation refinement of the method is presented in section V. We introduce our speculative chains method in section VI, explain the case study used for testing in section VII and review the practical results in section VIII. Section IX concludes the paper.

II. Markov Chain Monte Carlo

Markov Chain Monte Carlo is a computationally expensive non-deterministic iterative technique for sampling from a probability distribution that cannot easily be sampled from directly. Instead, a Markov Chain is constructed that has a stationary distribution equal to the desired distribution. We then sample from the Markov Chain, and treat the results as samples from our desired distribution. For a detailed examination of the MCMC method the reader is referred to [8]. Here we provide a summary of what the algorithm does in practise, excluding much of the theoretical backing.

At each iteration a transition is proposed to move the Markov Chain from state x to some state x', normally by making small alterations to x. The probability of applying this proposed move is calculated by a transition kernel constructed in such a way that the stationary distribution of the Markov Chain is the desired distribution. Such kernels produce the probability for advancing the chain to state x' from x based on how well x' fits with the *prior* knowledge (what properties the target configuration is expected to have) and the *likelihood* of x' considering the actual data available. The basic Metropolis-Hastings transition kernel can be expressed as

$$\alpha = \frac{\text{prior}(x')}{\text{prior}(x)} \times \frac{\text{likelihood}(x'|f)}{\text{likelihood}(x|f)} \times \frac{p(x', x)}{p(x, x')} \qquad (1)$$

where $p(x', x)$ is the probability of proposing the move from state x' to x. Transitions that appear to be favourable compared to the current state of the chain have $\alpha > 1$ and are accepted unconditionally, whilst moves to apparently worse states will be accepted with probability α. Once the move/transition has been either accepted (causing a state change) or rejected the next iteration begins. MCMC can be run for as many iterations as are required, the conventional use is to keep taking samples of the chain's state at regular intervals after an initial burn-in period to allow the chain to reach equilibrium. Depending on the needs of the application these samples can be then be compared to determine the most probable (most frequently occurring) characteristics amongst the samples. In some applications (typically those dealing with high-dimensional states, such as for image processing problems) a single sample from a chain that has reached equilibrium (converged) may be sufficient. Determining when a chain has converged (and therefore may be sampled) is an unsolved problem beyond the scope of this paper.

This paper concerns the fine-grained parallelisation of MCMC applications where the initial burn-in time is the most time-consuming period. Obtaining many samples is embarrassingly parallel as multiple chains with different initial models can be run on multiple computers. Samples from all the chains are simply grouped [9], not only reducing the time to obtain a fixed number of samples but also reducing the chance that all the samples will occur in local rather than global optima, since the chains will be starting from different positions in the state-space. However, running multiple chains does not change the initial burn-in time (waiting for the chains to move from their initial models to achieving equilibrium around optimal states), which for complicated and high-dimensional problems may be considerable.

III. Related work

The conventional approach to reducing the runtime of MCMC applications is to improve the rate of convergence so that fewer iterations are required. The main parallel technique is called Metropolis-Coupled MCMC (termed $(MC)^3$) [10], [11], where multiple MCMC chains are performed simultaneously. One chain is considered 'cold', and its parameters are set as normal. The other chains are considered 'hot', and will be more likely to accept proposed moves. These hot chains will explore the state-space faster than the cold chain as they are more likely to make apparently unfavourable transitions, however for the same reason they are less likely to remain at near-optimal solutions. Whilst samples are only ever taken from the cold chain, the states of the chains are periodically swapped, subject to a modified Metropolis-Hastings test. This allows the cold chain to make the occasional large jump across the state-space to avoid or escape local optima, avoiding the iterations where the chain would otherwise have been 'stuck' there. This differs from the work in this paper, where we seek to reduce the real-time required for MCMC without altering the path of the chain.

An alternative but complimentary method to $(MC)^3$ termed 'speculative moves' is detailed in [7]. This mechanism operates by speculatively considering multiple independent iterations simultaneously but then making the final decision whether to enact the state changes proposed in each iteration sequentially. Although by definition a Markov chain consists of a strictly sequential series of state changes, each MCMC iteration will not necessary result in a state change. In each iteration a state transition (move) is proposed but applied subject to the transition kernel. Moves that fail this test do not modify the chain's state so (with hindsight) need not have been

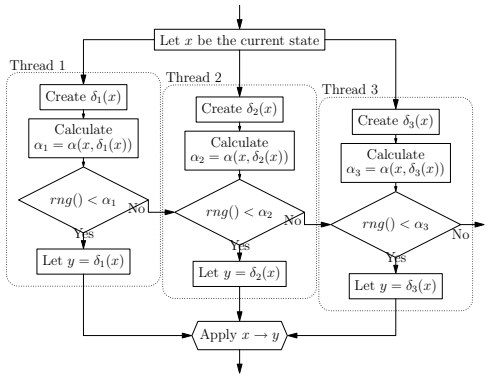

Fig. 1. Speculative move enabled program cycle.

evaluated. Consider a move 'A'. It is not possible to determine whether 'A' will be accepted without evaluating its effect on the current state's posterior probability, but we can assume it will be rejected and consider a backup move 'B' in a separate thread of execution whilst waiting for 'A' to be evaluated (see fig. 1). If 'A' is accepted the backup move 'B' - whether accepted or rejected - must be discarded as it was based upon a now supplanted chain state. If 'A' is rejected control will pass to 'B', saving much of the real-time spent considering 'A' had 'A' and 'B' been evaluated sequentially. Of course, we may have as many concurrent threads as desired, so control may pass to move 'C' if 'B' is rejected, then 'D', 'E', and so on. Obviously for there to be any reduction in runtime each thread must be executed on a separate processor or processor core. Interleaved threads will result in a net slowdown as the execution of speculative moves (that may or may not count towards the chain's iteration count) will delay the execution of 'normal' moves (that certainly will count).

As explained in [7], when using speculative moves the theoretical number of steps S (each step considering n iterations in parallel) required to perform I iterations of some MCMC application when the probability of any single move being rejected is p_r, can be expressed as

$$S_n = I\frac{1-p_r}{1-p_r^n} \tag{2}$$

Assuming no multithreading overhead, each step will take the same length of time as a single MCMC iteration, thus the program's runtime will be τS_n where τ is the average time required to perform a move. The value for p_r is not chosen directly, but as a consequence of tuning the MCMC program to give an acceptable rate of convergence. In practical applications a p_r value of around 0.75 is typical. When using speculative moves with two threads this allows a possible runtime reduction of 43%, or of 63% using four threads. In practice results close to these are obtainable [7] when using multicore architectures (for the minimum possible inter-process communication time) for applications with a comparatively high value for τ such that the interprocess communication time is insignificant compared to the time spent computing each iteration.

IV. Speculative Move Considerations

The speculative move method as expressed in [7] (and summarised above) assumes that a single value for mean processing time per move (τ) is adequate, and for the example simulations considered in that paper this is correct. However, there are applications where there may be substantial yet predictable variations in the time taken to process different types of move. Consider situations where the model being constructed contains composite structures such as trees (for example in the mapping of vascular trees as in [5], [12]). There may be moves that operate on individual features over small areas of the image (such as fine tuning a single node in the tree) and operations that modify large composite structures spread across large portions of the image. Even without composite structures there may be moves with effects that are highly localised (thus cheap to compute the change to the likelihood and prior terms should the move be applied) and others that modify variables with a non-localised effect forcing a (computationally expensive) complete recalculation of the prior and likelihood terms.

Instead of a single mean move time τ for all moves, let us consider a situation where we have a set $\mathbf{M_f}$ of moves that can be processed rapidly in time τ_f and set $\mathbf{M_s}$ that requires τ_s time to process, where $\tau_f \ll \tau_s$. For example, moves of set $\mathbf{M_f}$ will cause small alterations whose effect on the prior and likelihood terms can easy be calculated, whilst the moves of $\mathbf{M_s}$ result in more dramatic changes that require extensive or complete recalculations of the prior and likelihood terms. When using the *speculative move* mechanism as described in the preceding section the presence of set $\mathbf{M_s}$ moves amongst $\mathbf{M_f}$ moves causes inefficient processor utilisation. Consider one MCMC step with n threads. If at least one thread considers an $\mathbf{M_s}$ move, any thread that considers a $\mathbf{M_f}$ move must wait idle for $\tau_s - \tau_f$ whilst the $\mathbf{M_s}$ move completes processing. If the probability of any single MCMC iteration being a $\mathbf{M_f}$ move is q_f then the probability of a speculative move step taking time τ_s is $1 - q_f^n$ thus each step will on average take

$$\tau_f q_f^n + \tau_s(1 - q_f^n) \tag{3}$$

Combining this with eq. (2), the expected number of steps required, gives us a new expression for the predicted runtime for I iterations.

$$T = I\left(\tau_f q_f^n + \tau_s(1 - q_f^n)\right)\frac{1-p_r}{1-p_r^n} \tag{4}$$

Figure 2 shows this plotted for varying q_f with common values of n (1,2,4,8), $p_r = 0.75$, and each long moves taking five times the processing time of a typical short move. The y-axis is the normalised runtime, such that '1' is the time taken for the sequential program to complete a fixed number of iterations with no $\mathbf{M_s}$ moves being proposed. The benefit of speculative moves (relative to equivalent sequential runtime) is of course identical if all moves performed are from the same set ($\mathbf{M_f}$ or $\mathbf{M_s}$, corresponding to $1 - q_f = 0$ and $1 - q_f = 1$ respectively). For values of q_f in-between, the runtime-reducing effect of speculative moves is impaired, although the

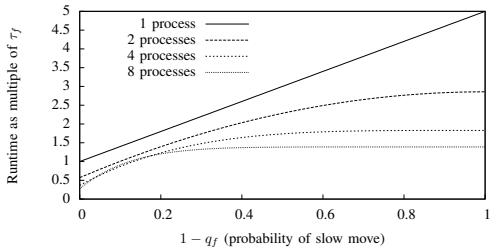

Fig. 2. Impact of $\mathbf{M_s}$ moves on spec. move runtime, $p_r = 0.75$, $\tau_s = 5\tau_f$

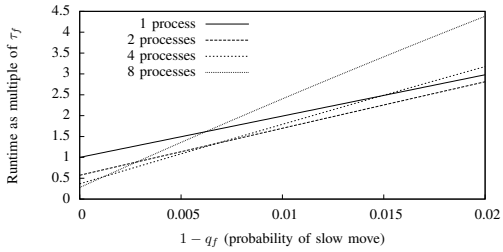

Fig. 3. Impact of $\mathbf{M_s}$ moves on spec. move runtime, $p_r = 0.75$, $\tau_s = 100\tau_f$

speculative move implementations do not become slower than the sequential version. The benefit provided by multithreading is thus reduced; instead of providing a runtime reduction of $\approx 43\%$ the two threaded version only reduces runtime by $\approx 22\%$ when 20% of moves are $\mathbf{M_s}$. As $\mathbf{M_s}$ moves become the norm ($q_f \to 0$) the benefit of speculative moves is of course restored, although always at a net increase in runtime.

In the more extreme case of fig. 3, where $\mathbf{M_s}$ moves take the time of 100 $\mathbf{M_f}$ moves, the benefits of speculative moves are lost though the presence of comparatively few $\mathbf{M_s}$ moves (note that the scale along the x-axis only goes up to a $\mathbf{M_s}$ move proposal probability of 0.02). Obviously the presence of $\mathbf{M_s}$ moves is going to increase the runtime, but when speculative moves are used the effect is disproportionately large for certain values of q_f, as is clear by the sharp curvature of speculative move lines. If more than 1.5% of moves are of the long duration variety all benefits of four-thread speculative moves are lost, the method becoming a hindrance until at least 25% of moves are from $\mathbf{M_s}$.

V. Improving Speculative Moves

The presence of $\mathbf{M_s}$ moves has a detrimental effect when using speculative moves because they impair thread utilisation. In each program cycle involving a $\mathbf{M_s}$ move, threads performing a $\mathbf{M_f}$ move are left idle whilst they wait for the slow move to complete. The "naive" implementation of speculative moves presented earlier guarantees that all speculative moves will be employed at each loop around the program cycle by synchronising the threads (waiting for all move calculations to complete) before starting the next set of move proposals. The threads are used eagerly, regardless of the circumstances. The alternative is to use the threads lazily, a thread will only be used for a step if the thread is available when we need it.

Under this revised implementation, if a proposed move is rejected we will wait for the speculative move(s) to make decisions and act accordingly (as before). However, when a proposed move is accepted any additional speculative move threads are messaged to be cancelled, then the primary thread immediately begins work on the follow-up move. When a new speculative move needs to be processed, any threads that are still working on (or are in the process of cancelling) a preceding speculative move are ignored and for that program cycle fewer speculative moves than normal are used. Speculative moves are considered on a best-effort basis, so will not be employed at the expense of delaying work on moves that are guaranteed to count towards the total number of MCMC iterations performed. Since the maximum number of speculative moves may not be utilised if one or more threads are busy, more steps (loops round the program cycle) will be required to obtain the same number of MCMC iterations, however the average time per step will be decreased as it will no longer be necessary to wait for invalidated $\mathbf{M_s}$ moves to complete.

A consequence of allowing the effective number of threads performing speculative moves to vary is the possibility that all threads performing speculative moves may become 'blocked' performing $\mathbf{M_s}$ moves, leaving just the primary thread to perform work and resulting in near-sequential runtime. This will only become an issue if $\mathbf{M_s}$ moves are proposed faster than they can be cleared from a speculative thread. Assuming that any move once started must run to completion on its thread, to avoid all speculative threads becoming blocked we must have

$$\begin{aligned} (1 - q_f)^n &< \frac{\tau_f}{\tau_s} \\ q_f &> \sqrt[n]{1 - \frac{\tau_f}{\tau_s}} \end{aligned} \tag{5}$$

Such a constraint is unwelcome ($< 10\%$ slow moves when a slow move is 5 times the length of the fast one, $< 0.5\%$ of moves if slow moves take the time of 100 fast moves), so implementers are encouraged to consider the cancelling of move calculations as they design the move proposal and prior/likelihood implementation. Killing and restarting a thread is not an option due to the risk of deadlocks, so polling some 'abort' flag at regular intervals through the long-running calculations will be necessary. Naturally to ensure thread safety, access to this thread must be synchronised (reads/writes controlled by a pthread mutex). Due to the potential expense of the many mutex operations per move, the provision for promptly aborting consideration of a cancelled move should be provided only when most required, such as for $\mathbf{M_s}$ moves.

VI. Speculative Chains

In the preceding section we have reduced or eliminated the impact of invalid $\mathbf{M_s}$ moves on the program runtime. We will now address the bottleneck caused by necessary $\mathbf{M_s}$ moves by extending the speculative move philosophy. Let there be n threads labelled $1..n$ in order of priority, thread 1 being the primary thread and threads $2..n$ the threads performing speculative moves (in descending order of preference). If thread i is considering a move from $\mathbf{M_s}$, all threads $> i$

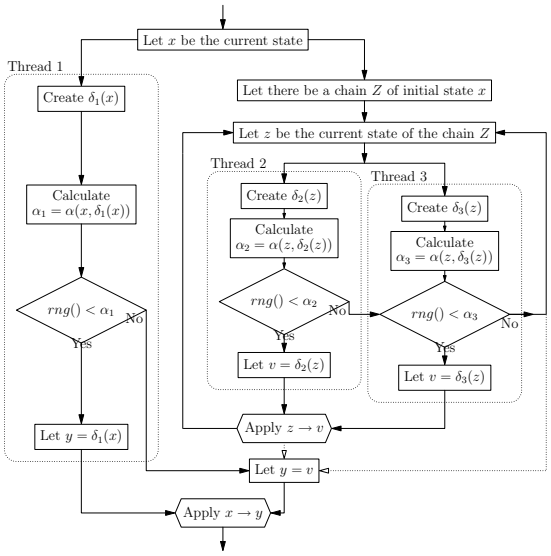

Fig. 4. Example program cycle using a speculative chain. This is the program cycle that will occur when thread 1 performs a long duration ($\mathbf{M_s}$) move.

that consider a $\mathbf{M_f}$ move will be idle for $\tau_s - \tau_f$ whilst they await a decision to be made on i's move. If the i'th thread's move is rejected this idle time is a waste. Should i's move be accepted, it is irrelevant - their results will be discarded as the i'th thread will enact a state-change invalidating any other speculative moves considered in that step.

To avoid unnecessary idle time whenever a thread i performs a $\mathbf{M_s}$ move we perform a speculative *chain* on thread $i+1$. Instead of using this thread to propose and test a single move, we create a temporary clone of the current chain state and permit multiple moves to be considered and possibly accepted on this copy. Once thread i has completed its move we can determine the fate of this speculative chain. If thread i's move is accepted the speculative chain will be discarded, otherwise its state will be used as the new state of the primary chain and we can return to the normal speculative move program cycle until the next $\mathbf{M_s}$ move is encountered. The speculative chain need not be confined to a single thread, if we still have threads $> i+1$ to spare then they may be utilised to perform speculative moves for the speculative chain (possibly allowing further speculative chains to be formed). This is illustrated in fig. 4, where a single program cycle is shown for the case when the first thread (1) conducts a $\mathbf{M_s}$ move.

The use of a speculative chain is only desirable when a $\mathbf{M_s}$ move is being considered in an earlier thread in the same step, and even then only if τ_s is long enough to justify the potentially substantial overhead involved in cloning the state of the primary chain. This chain-state cloning may be implemented in one of three ways: immediate, deferred, or virtual duplication. With immediate duplication. The master chain state is copied in memory upon the creation of the speculative chain. Whilst expensive for large or complicated models, this is the simplest to implement. Deferred duplication delays copying the master chain state until the speculative chain first accepts a move. This is cheap if it is likely the speculative chain will not find an acceptable move before its preceding thread completes processing of the $\mathbf{M_s}$ move, i.e. p_r is high and $\frac{\tau_s}{\tau_f}$ is small. The downside is the less predictable processing time and the added complexity: whilst the chain state cloning procedure is underway the speculative chain may receive an abort request, and/or the master chain state may be changed due to the action of an accepted speculative move (applied to the master chain). Immediate duplication will be preferable if it is probable the speculative chain will accept at least one move. Virtual duplication avoids the overhead of copying the master chain state, instead a record is kept of each of the accepted state changes. New move proposals consider the original chain as seen through a 'filter' that takes account of the sequence of accepted statechanges. Instead of a large upfront processing cost, the overhead is applied to each iteration of the speculative chain, with the overhead increasing with each move that is accepted. Whilst the hardest to implement, virtual duplication is potentially the most efficient when dealing with large models that are expensive to copy directly (i.e. phylogenies that may be megabytes in size [10]), provided the speculative chain does not accept too many moves (as the overhead per move increases with each extra accepted move in the speculative chain).

Whilst the overhead of a speculative chain may be substantial and (as with speculative moves) runtime benefits will only be obtained if the primary move is rejected, there is the potential for performing up to

$$\frac{\tau_s}{\tau_f q_f + \tau_s(1-q_f)} \tag{6}$$

sequential iterations within the speculative chain. This may be increased further if there are remaining unused processors, as the speculative chain may then itself utilise speculative moves and chains.

Using the extensions to the speculative moves mechanism discussed above, for the case where there are two available processors there are four possibilities to be considered:

1) The primary move is from $\mathbf{M_f}$ and accepted (probability $q_f(1-p_r)$). The step performs one iteration and takes τ_f (assuming the overhead of aborting a move is negligible).
2) The primary move is from $\mathbf{M_f}$ and rejected (probability $q_f p_r$). Two iterations occur, but the time is dependant on the type of move considered speculatively. On average, it will take $\tau_f q_f + \tau_s(1-q_f)$.
3) A primary move from $\mathbf{M_s}$ that is accepted (probability $(1-q_f)(1-p_r)$) will naturally take τ_s time to perform the single MCMC iteration.
4) A rejected $\mathbf{M_s}$ primary move ($(1-q_f)p_r$) will still take τ_s time, but performs the number of iterations expressed in equation (6) $+1$.

Although we can use this information to derive a formula for the predicted runtime using speculative chains methodology, it is simpler (particularly when dealing with the 4, 8, or more threaded versions of the problem) to construct a simulator to loop through and sum up the expected runtime of each program cycle, accounting for the presence of speculative

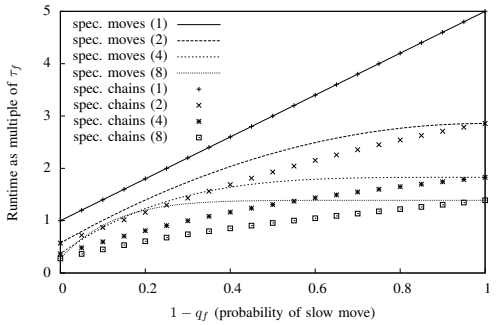

Fig. 5. Impact of $\mathbf{M_s}$ moves on spec. chain runtime, $p_r = 0.75$, $\tau_s = 5\tau_f$

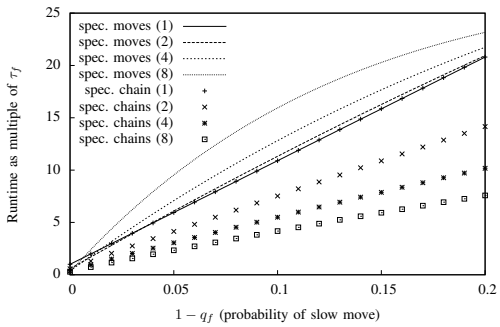

Fig. 6. Impact of $\mathbf{M_s}$ moves on spec. chain runtime, $p_r = 0.75$, $\tau_s = 100\tau_f$

moves and threads. This simulator, implemented in Java and running on a Q6600 machine is capable of simulating 1.5×10^6 4-threaded program cycles a second. Some results from this simulator are shown in fig. 5 and fig. 6. The speculative move lines are the results obtained from eq. (4) whilst the speculative chain results are those from the simulator, the numbers in brackets is the number of processors/processing cores available. Along the x axis is the probability/frequency by which $\mathbf{M_s}$ moves are proposed. The y axis shows the normalised runtime, a value of '1' is the time the sequential code would take if there are no $\mathbf{M_s}$ moves proposed. The same assumptions used in the formulaic predictions of the previous section apply, the multithreading overhead is considered negligible so only move calculations are considered time consuming. Additionally we assume that long-running moves may be aborted with negligible cost, and that a single value for the p_r holds for both $\mathbf{M_s}$ and $\mathbf{M_f}$ moves (in this example $p_r = 0.75$).

As shown by fig. 5, the use of speculative chains can provide a substantial performance improvement over speculative moves if there exist moves that (predictably) take only 5 times longer than normal. If the difference between the $\mathbf{M_s}$ and $\mathbf{M_f}$ moves is greater (such as by x100 as in fig. 6 - note the different scale along the x-axis) it takes only a small percentage of moves to be in the $\mathbf{M_s}$ set for the speculative chains method to yield substantial results. In both cases the curves for simulated speculative chains are much flatter, retaining most of the benefit of speculative moves almost irrespective of the proportion of $\mathbf{M_s}$ moves (though note that as with the speculative moves predictions, these represent the upper bound on performance improvements).

VII. Case study: artifact recognition

An example application of MCMC in the context of image processing is the identification of features/artifacts in an image. For instance, the counting of tree crowns in from satellite images [4], tracking heads in a crowd or counting stained cells in slides of a tissue sample. We abstract this problem to recognising and counting artifacts (in this case circles) of high colour intensity in an image, if necessary filtering the image first to emphasise the colour of interest. This filtered image can then be used to produce a model for the original image - a list of artifacts i.e. circles defined by their coordinates and radii. A random configuration is generated and used as the initial state of the Markov Chain. At each iteration a type of alteration is chosen at random. The possible alterations are the artifact addition or deletion, merging two artifacts together, splitting an artifact into two, and altering the position or properties (i.e. radius) of an existing artifact. A *move proposal* (possible state transition) is then generated that implements an alteration of that type, randomly choosing the artifact(s) to alter and the magnitude of that alteration. The probability of accepting this move is generated by a Metropolis-Hastings transition kernel constructed using Bayesian inference.

Two terms, the *prior* and *likelihood*, are calculated to evaluate the configuration that would be created by this move proposal. The *prior* term evaluates how well the proposed configuration matches the expected artifact properties, in this case the distribution and size of the circles and the degree to which overlap is tolerated. The *likelihood* of the proposed configuration is obtained by comparing the proposed artifacts against the filtered image. Together the prior and likelihood terms make up the *posterior* probability of the proposed configuration, which is compared with the posterior probability of the existing state using a reversible-jump Metropolis-Hastings transition kernel [13]. Put simply, whilst the prior and likelihood probabilities cannot be expressed exactly, the ratio between the posterior probabilities of the current and proposed configurations can be calculated and used to give a probability for accepting the state change.

VIII. Results

To obtain the time-per-move characteristics required for testing the speculative chains method, an alternative 'alter artifact position' move was introduced that performed a more detailed calculation of the prior and likelihood terms, taking two to three times the time of simpler (but less accurate) moves. When a larger disparity between move times was required, a loop was inserted to force the likelihood calculations to be repeated multiple times. In the field, such move consideration time differences will be down to composite moves effecting large portions of the model (for instance, remove or modify an entire connected component of features as in 'delete tree' moves in [5]), the presence of more advanced logic in certain moves (for instance, a guided placement of new features rather than proposed new features being located entirely at random), or simply more complex prior/likelihood calculations than those for the other types of move.

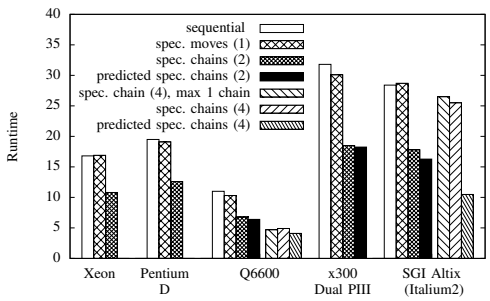

Fig. 7. Comparison of speculative chains employed on different architectures. $p_r = 0.75$, $q_f = 0.999$, $\tau_s = 5\tau_f$

The benefits of speculative chains can be seen in fig. 7. This shows the runtime of the case study program working on autogenerated 1024x1024 images containing 300 circles, where $p_r = 0.75$ and $q_f = 0.999$ (0.1% of moves are from the $\mathbf{M_s}$ set). The hardware used was a IBM xSeries 330 Dual-Pentium III Processor, an Intel Xeon Dual-Processor, an Intel Pentium-D (dual core), an Intel Core2 Quad Q6600 (2x dual-core dies), and an 56-Itanium2 processor SGI Altix. Despite less than 0.1% of moves being from $\mathbf{M_s}$ and τ_s being only 3-5 times τ_f, the speculative moves mechanism is rendered ineffective on all platforms, taking nearly as long (if not longer, in the case of the Altix) than the normal sequential execution. Supplementing speculative moves with a single speculative chain whenever an $\mathbf{M_s}$ move is considered on the primary thread substantially improves performance across all architectures that were tested, giving runtime reductions in the range of 35-42%. When four threads are available, just using speculative moves yields the same results as speculative moves with two threads (not shown in fig. 7). Speculative chains were tested using four threads in two different ways. Firstly allowing at most one speculative chain to be active at any one time to limit the number of state-cloning procedures that would be performed, and secondly allowing as many speculative chains as will fit (in this case, three). Results were mixed, whilst the Q6600 yielded results close to those predicted (achieving a reduction of 57%), the SGI Altix seems ill-suited to this 4-threaded speculative chains parallelisation as the results produced were only marginally better than the sequential program, obtaining a runtime reduction of only 10%. This is due to a combination of the state-cloning implementation used (the memory-intensive immediate duplication) and the architecture of the Altix. Processing nodes are arranged in pairs with shared cache but communication between non-paired nodes is done through main memory. Since exclusive access to the Altix was not available, memory usage was restricted thus state-cloning was a bottleneck dramatically impairing performance when many main-memory accesses were required, this could be somewhat improved by using a more memory efficient means of state-cloning such as virtual duplication. The problem did not arise when using only two threads as they could be performed on paired processing nodes and communication/state-cloning performed through the shared cache.

TABLE I
COMPARISON BETWEEN FAST AND SLOW MOVES. 300 CIRCLES IN A 1024X1024 IMAGE, $p_r = 0.75$, 0.1% OF MOVES FROM $\mathbf{M_s}$

	τ_f	τ_s	$\frac{\tau_s}{\tau_f}$
x330 Dual-Processor	1.59×10^{-3}	6.75×10^{-3}	4.23
Q6600 (2x dual core)	5.62×10^{-4}	2.53×10^{-3}	4.5
Q6600⋆ (2x dual core)	4.29×10^{-4}	2.15×10^{-3}	5.02
Altix (56 processor)	1.42×10^{-3}	4.18×10^{-3}	2.94

TABLE II
THE % OF THE POTENTIAL (THEORETICAL) REDUCTION IN RUNTIME ACHIEVED. $p_r = 0.75$, 0.1% OF MOVES FROM $\mathbf{M_s}$.

	2 threads	4 threads
x330 Dual-Processor	98	-
Q6600 (2x dual core)	91	88
Q6600⋆ (2x dual core)	89	70
Altix (56 processor)	87	16

The predicted values in fig. 7 were obtained using the simulator from section VI and measurements from the time spent performing each of the different types of move. Those move-time measurements are listed in table I.

The degree to which the different architectures achieved their potential runtime improvement is displayed in table II. Unlike the results for speculative moves [7] the best performers are not grouped by their multithreading capabilities (dual-processor/dual-core/quad-core). This is because the overhead imposed by multithreading is less significant compared to the timeframe concerned (a single chain will last three to five times longer than an ordinary move in this example), the main cause of overhead is the workload involved in cloning the MCMC chain's state.

Table I shows there is an added complication when comparing architectures on which to use speculative chains. Not only will the time per (sequential) MCMC iteration change depending on the hardware, but the ratio $\frac{\tau_s}{\tau_f}$ will as well. This difference in scaling between fast and slow moves is caused by the way the additional workload present in $\mathbf{M_s}$ moves is processed on different hardware and software environments, specifically differences in compiler optimisations, kernel efficiency, memory/cache latency, and build-in hardware optimisations (such as pipelining within the processor).

To illustrate some of the problems that can arise, tables I and II contain results for two different Q6600 machines. The data labelled Q6600 is from a machine running Linux 2.6.18-36, and with the test program compiled using the GNU compiler GCC version 4.1.1. Q6600⋆ is a separate machine running the Linux 2.6.27-11-server and using GCC version 4.3.2. Unsurprisingly the machine with the newer compiler, kernel version, and server-optimised kernel performs the move computations faster, although from table II it implemented speculative moves less efficiently than its 'slower' equivalent, implying that there is a bottleneck whose rate of progress was less effected by the differing software configuration. A contributing factor to this bottleneck is the higher $\frac{\tau_s}{\tau_f}$ ratio of the Q6600⋆.

Figure 8 shows the processing of the same data as fig. 7

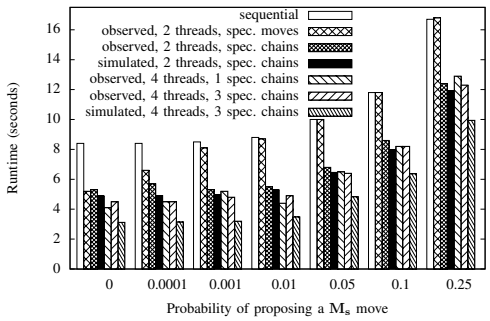

Fig. 8. Altering $\mathbf{M_s}$ move proposal probability. $p_r = 0.75$, $\tau_s = 5\tau_f$.

but for varying $p_r^{\mathbf{M_s}}$ (the probability of any one move being a member of $\mathbf{M_s}$) on just the Q6600 machine, $\tau_s \approx 5\tau_f \approx 2ms$ (20 000 MCMC iterations total). If no $\mathbf{M_s}$ moves are present performance improvements from speculative moves are as would be expected from eq. (2). When $\mathbf{M_s}$ moves are proposed the benefit of speculative moves is lost (in this case when the percentage of $\mathbf{M_s}$ moves is somewhere in the range of $0.001\% \rightarrow 0.01\%$) and using speculative moves yields no performance improvement over the sequential program. Allowing a single speculative chain to be used (instead of a speculative move) when a $\mathbf{M_s}$ move is proposed allows the performance improvement from using speculative moves to be maintained despite the presence of $\mathbf{M_s}$ moves. Under the conditions used in this test the use of a single thread to perform speculative move/chains results in a reduction of runtime by around 33% from the sequential (or solely speculative-move-enabled) program, in line with predictions.

When four threads were available to the program, using just speculative moves yielded comparable results to using two threads with only speculative moves, the delay caused by the presence of $\mathbf{M_s}$ moves being limited to around that of the sequential implementation due to the ability to cancel such moves (the two thread and four thread speculative move runtimes exceed the sequential implementation due to overheads of multithreading and synchronisation, and because $\mathbf{M_s}$ cannot be truly be cancelled 'immediately'). With the addition of speculative chains runtime was reduced to, at best, that of the 2-thread speculative move implementation regardless of how many speculative chains were permitted to operate simultaneously, falling short of predictions as $\mathbf{M_s}$ moves become more frequent. This shortfall can be at least partially attributed to the overhead in cloning the state for each speculative chain, as mentioned earlier, the case study application used only the immediate duplication method for copying the chain state when starting new chains, whilst the predictions were made by a simulator assuming negligible overhead in this regard (i.e. virtual duplication).

IX. Summary

In this paper the basic speculative moves method presented in [7] has been refined and extended to accommodate MCMC iterations of variable (yet predictable) realtime durations. When there is significant variation in the processing duration between different types of iteration, there may be sufficient time to speculatively consider a chain of iterations whilst a single, long duration move is considered. Whilst ignoring variations in processing duration between iteration types may result in a speculative move implementation that is substantially slower than the original sequential version, proper use of speculative chains can produce reductions in runtime exceeding the most optimistic predictions of speculative moves.

Speculative moves and speculative chains are transparent to the statistical algorithm in use, although the implementation is different, the end result is indistinguishable from a traditional sequential implementation. These methods may be used in conjunction with conventional methods for reducing MCMC application runtime by improving convergence i.e. $(\text{MC})^3$. Since the speculative methods can only be applied where interprocessor communication overheads are small compared to the iteration time, a natural division is to use $(\text{MC})^3$ across multiple machines in a cluster, then speculative moves/chains on each individual multicore/multiprocessor machine.

References

[1] S. Li, D. K. Pearl, and H. Doss, "Phylogenetic tree construction using Markov chain Monte Carlo," *Journal of the American Statistical Association*, 1999.

[2] M. Bonamente, M. K. Joy, J. E. Carlstrom, E. D. Reese, and S. J. LaRoque, "Markov chain Monte Carlo joint analysis of Chandra X-ray imaging spectroscopy and sunyaevzel'dovich effect data," *The Astrophysical Journal*, vol. 614, no. 1, pp. 56–63, 2004.

[3] M. Johannes and N. Polson, "MCMC methods for financial econometrics," in *Handbook of Financial Econometrics*. North-Holland. Forthcoming, 2002.

[4] G. Perrin, X. Descombes, and J. Zerubia, "A marked point process model for tree crown extraction in plantations," in *IEEE International Conference on Image Processing*, vol. 1, 2005, pp. 661–4.

[5] D. C. K. Fan, "Bayesian inference of vascular structure from retinal images," Ph.D. dissertation, University of Warwick, May 2006.

[6] E. Thonnes, A. H. Bhalerao, W. Kendall, and R. Wilson, "A Bayesian approach to inferring vascular tree structure from 2D imagery," in *International Conference on Image Processing*, vol. 2, 2002, pp. 937–940.

[7] J. M. R. Byrd, S. A. Jarvis, and A. H. Bhalerao, "Reducing the run-time of MCMC programs by multithreading on SMP architectures," in *IEEE International Symposium on Parallel and Distributed Systems (IPDPS)*, 2008.

[8] P. J. Green, *Practical Markov Chain Monte Carlo*. Chapman and Hall, 1994.

[9] J. S. Rosenthal, "Parallel computing and Monte Carlo algorithms," *Far East Journal of Theoretical Statistics*, vol. 4, pp. 207–236, 2000.

[10] G. Altekar, S. Dwarkadas, J. P. Huelsenbeck, and F. Ronquist, "Parallel Metropolis-Coupled Markov chain Monte Carlo for Bayesian Phylogenetic Inference," Department of Computer Science, University of Rochester, Tech. Rep. 784, July 2002.

[11] M. Harkness and P. Green, "Parallel chains, delayed rejection and reversible jump MCMC for object recognition," in *British Machine Vision Conference*, 2000.

[12] E. Thonnes, A. H. Bhalearo, W. S. Kendall, and R. Wilson, "Bayesian analysis of vascular structure," in *Stochastic Geometry, Biological Structure and Images*, R. G. Ackroyd, K. V. Mardia, and M. J. Langdon, Eds., 2003, pp. 115–118.

[13] P. J. Green, "Reversible jump markov chain monte carlo computation and bayesian model determination," *Biometrika*, vol. 82, pp. 711–732, 1995.