

# Reducing the Run-time of MCMC Programs by Multithreading on SMP Architectures

Jonathan M. R. Byrd, Stephen A. Jarvis and Abhir H. Bhalerao

University of Warwick

Department of Computer Science

Coventry, CV4 7AL, UK

{J.M.R.Byrd, Stephen.Jarvis, 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 found a wide variety of applications in fields including computational biology and physics, financial econometrics, machine learning and image processing.*

*This paper presents a new method for reducing the runtime of Markov Chain Monte Carlo simulations by using SMP machines to speculatively perform iterations in parallel, reducing the runtime of MCMC programs whilst producing statistically identical results to conventional sequential implementations. We calculate the theoretical reduction in runtime that may be achieved using our technique under perfect conditions, and test and compare the method on a selection of multi-core and multi-processor architectures. Experiments are presented that show reductions in runtime of 35% using two cores and 55% using four cores.*

## 1. Introduction

Markov Chain Monte Carlo (MCMC) is a computational intensive technique for sampling from a (typically very large) probability distribution. They are most commonly applied to calculating multi-dimensional integrals, and as such have numerous applications in Bayesian statistics, computational physics and computational biology. Notable examples include constructing phylogenetic trees [10, 12], spectral modelling of X-ray data from the Chandra X-ray satellite [2], and for calculating financial econometrics [11].

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 arranged in a regular pattern [14]. 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 a measure of variability in solutions it provides.

Monte Carlo applications are generally considered embarrassingly parallel [15], since samples can be obtained twice as fast by running the problem on two independent machines. This also applies for Markov Chain Monte Carlo, provided a sufficient burn-in time has elapsed and a sufficient number of distributed samples are taken. Unfortunately for high-dimensional problems for which MCMC is best suited, the burn in time required for getting good samples can be considerable. When dealing with very large state-spaces and/or complicated compound states (such as searching for features in an image) it can take a long time for a MCMC simulation to converge on a satisfactory model, both in terms of the number of iterations required and the complexity of the calculations occurring in each iteration. As an example, the mapping of vascular trees in retinal images as detailed in [3, 16] 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.

If modes are not expected to be radically different, duplicating the simulation will not substantially reduce runtime as the time required for convergence will dominate over the time collecting samples. Statistical techniques already exist for improving the rate of convergence, indeed most current optimisation and/or parallelisation strategies take this

approach. The motivation for the work presented in this paper is to find methods of reducing the runtimes of MCMC applications by focusing on the implementation of MCMC rather than the statistical algorithm.

The contributions of this paper are threefold:

- We propose a new method (termed ‘speculative moves’) of implementing Markov Chain Monte Carlo algorithms to take advantage of multi-core and multi-processor machines.
- We fully implement and test this method on five different machine architectures and demonstrate the suitability of these architectures for this new approach.
- Finally, we provide a method for predicting the runtime of MCMC programs using our speculative moves approach, therefore providing: (i) increased certainty in real-world MCMC applications, (ii) a means of comparing alternative supporting architectures in terms of value for money and/or performance.

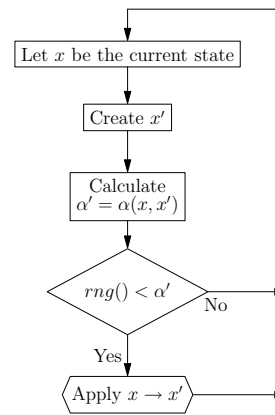
‘Speculative moves’ may be used alongside most existing parallelisation and optimisation techniques whilst leaving the MCMC algorithm untouched, and so may safely be used without fear of altering the results. Our method is designed to operate on the increasingly available multiprocessor and multicore architectures. As the technology improves (e.g. by increasing the number of processing cores that are placed onto a single die) the speculative moves method will yield greater runtime reductions over a wider range of applications.

The remainder of this paper is organised as follows. In section 2 we explain the MCMC method and the difficulties in parallelising it. Section 3 reviews the current forms of parallel MCMC. Our method of speculative moves is outlined in section 4, with the theoretical improvements possible calculated in section 5. We introduce the case study to which we applied speculative moves in section 6 and review the results in section 7. Section 8 concludes the paper.

## 2. Markov Chain Monte Carlo

Markov Chain Monte Carlo is a computationally expensive nondeterministic 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 [6]. Here we provide a summary of what the algorithm does in practise, excluding much of the theoretical backing.

At each iterations a transition is proposed to move the Markov Chain from state  $x$  to some state  $y$ , normally by



**Figure 1. Conventional Markov Chain Monte Carlo Program Cycle - one MCMC iteration is performed at each step of the cycle**

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. The construction of a suitable kernel is often surprisingly easy, and is frequently done by applying Bayesian inference [7]. Such kernels produce the probability for advancing the chain to state  $y$  from  $x$  based on how well  $y$  fits with the *prior* knowledge (what properties the target configuration is expected to have) and the *likelihood* of  $y$  considering the actual data available. Transitions that appear to be favourable compared to the current state of the chain have acceptance probabilities  $> 1$  and so are accepted unconditionally, whilst moves to apparently worse states will be accepted with some reduced probability. Once the move/transition has been either accepted (causing a state change) or rejected the next iteration begins. This program cycle is shown in figure 1. 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 will either be the subject of further processing or compared to identify the ‘best’ (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 of a chain that has reached equilibrium (converged) may be enough. Determining when a chain has converged (and therefore may be sampled) is an unsolved problem beyond the scope of this paper.

This paper concerns parallelising MCMC applications where the initial burn-in time is the most time-consuming period. Obtaining many samples is embarrassingly parallel as multiple chains can be run on multiple computers,

each using a different initial model but keeping all other factors the same. Samples from all the chains can be simply grouped [15], not only reducing the time to obtain a fixed number of samples but also reducing the chances that all the sample 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.

### 3. 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  $(MC)^3$ ) [1, 9], 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 whilst still converging on good solutions.

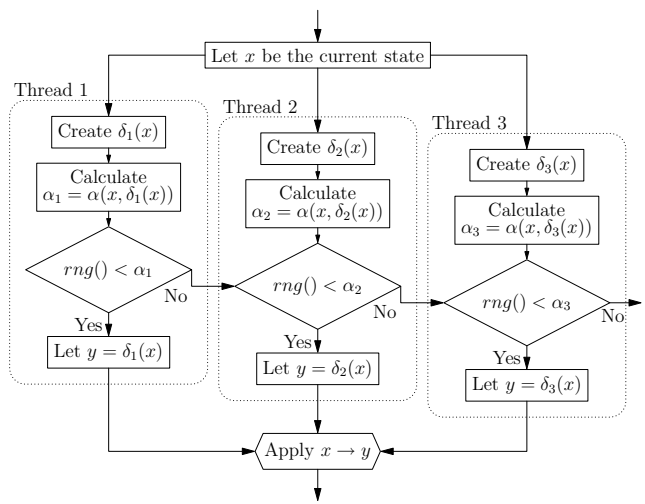
$(MC)^3$  differs from our work in that communication between chains is infrequent, thus the chains can be executed across networked computers. The aims are also very different -  $(MC)^3$  increases the mixing of the chain, improving the chances of discovering alternative solutions and helping avoid the simulation becoming stuck in local optima. Essentially it reduces the number of iterations required for the simulation to converge, whereas our method reduces the time required to perform a number of iterations. The two approaches will complement each other, particularly since  $(MC)^3$  permits its chains to be spread over multiple computers connected by a comparatively low speed interconnects.

The technique of delayed rejection MCMC, first proposed by [17] and then generalised in [8], seeks to reduce the probability of iterations that do not advance the state of the chain. If a move proposal is at first rejected, a second is attempted that may optionally depend upon the rejected move. This improves performance of the sampler, but at the cost of increased computation per iteration (at least, those iterations that initially reject a transition). Our method also targets rejected moves as a place for optimisation, but does

so through changes in implementation and parallel processing rather than by altering the statistical algorithm.

In some applications it is possible to split the input dataset or identify traits that can be considered and processed independently, as in the parallel phylogenetic inference presented in [4]. As with  $(MC)^3$  this method of parallelisation is coarse enough to work over a network, however it is very application specific. In the general case making such clean divisions in the input data or internal representation is not possible. For instance when processing images naively, bisecting the image and considering the two halves separately will lead to anomalies near the sub-image boundary and a loss of the statistical principals underpinning the MCMC methodology.

### 4. A new parallelising approach based on speculative moves



**Figure 2. Speculative Move Enabled Program Cycle.** In this case three potential moves are considered at each step of the program cycle. This translates to one, two or three MCMC iterations being performed, depending on whether the first and second potential moves are accepted or rejected.

Although by definition a Markov chain consists of a strictly sequential series of state changes, each MCMC iteration will not necessarily result in a state change. In each iteration (see figure 1) a state transition (move) is proposed but applied subject to the Metropolis-Hastings test. Moves that fail this test do not modify the chain’s state so (with hindsight) need not have been 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 figure 2). 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.

To be useful the speculative move must not compete with the initial move for processor cycles. In addition, the communication overhead for synchronising on the chain’s current state, starting the speculative moves and obtaining the result must be small compared to the processing time of each move. An SMP architecture is most likely to meet these criteria, though a small cluster might be used if the average time to consider a move is long enough. As many speculative moves may be considered as there are processors/processing cores available, although there will be diminishing returns as the probability of accepting the  $m$ th speculative move is  $(p_r)^{m-1}(1 - p_r)$  where  $p_r$  is the probability of rejecting any one move proposal.

Since speculative moves compress the time it takes to perform a number of iterations, the method will complement existing parallelisation that involves multiple chains to improve mixing or the rate of convergence (such as  $(MC)^3$  or simply starting multiple chains with different initial models), provided sufficient processors are available. As the other parallelisation methods tend to be less tightly coupled it is feasible for physically distinct computers to work on different chains, whilst each chain makes use of multiple cores/processors on its host computer for speculative moves.

## 5. Theoretical gains

When using the speculative move mechanism with  $n$  moves considered simultaneously, the program cycle consists of repeated ‘steps’ each performing the equivalent of between 1 and  $n$  iterations. The moves are considered in sequence, once one move has been accepted all subsequent moves considered in that step must be ignored.

Given that the average probability of a single arbitrary move being rejected is  $p_r$ , the probability of the  $i^{th}$  move in a step being accepted whilst all preceding moves are rejected is  $p_r^{i-1}(1 - p_r)$ . Such a step counts for  $i$  iterations. Including the case where all moves in a step are rejected (occurring with probability  $p_r^n$ ), the number of iterations ( $I$ ) performed by  $S_n$  steps (where  $n$  is the number of moves

considered in each step) can be expressed as

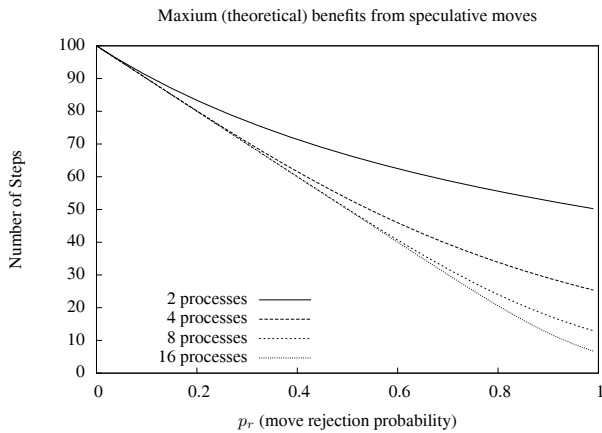
$$\begin{aligned}
 I &= S_n \left[ \sum_{i=1}^n i p_r^{i-1} (1 - p_r) + n p_r^n \right] \\
 I &= S_n \left[ \sum_{i=1}^n i p_r^{i-1} - \left( \sum_{i=1}^n i p_r^i - n p_r^n \right) \right] \\
 I &= S_n \left[ \sum_{i=1}^{n-1} (i+1) p_r^i + p_r^0 - \sum_{i=1}^{n-1} i p_r^i \right] \\
 I &= S_n \left[ \sum_{i=1}^{n-1} p_r^i + 1 \right] \\
 I &= S_n \left[ \frac{p_r(1 - p_r^{n-1})}{1 - p_r} + 1 \right] \\
 I &= S_n \frac{1 - p_r^n}{1 - p_r}
 \end{aligned}$$

Rearranging for  $S_n$

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

which is plotted in figure 3 for varying  $p_r$ . Assuming the time taken to apply an accepted move and the overhead imposed by multithreading are both negligible compared to the time required for move calculations, the time per step  $\approx$  time per iteration. Therefore figure 3 also shows the limits of how the runtime could potentially be reduced. For example, if 25% of moves in an MCMC simulation are accepted ( $p_r = 0.75$ ), 100 sequential iterations are equivalent to  $\approx 57$  steps for a two-threaded speculative move implementation or  $\approx 37$  steps on a four-threaded implementation. Four thread speculative moves could therefore at best reduce the runtime of a MCMC application accepting 25% of its moves by about 63%, while the two threaded version could achieve up to a 43% reduction.

In practise speedups of this order will not be achieved. Threads will not receive constant utilisation (as they are synchronised twice for each iteration) so may not be consistently scheduled on separate processors by the OS. For rapidly executing iterations the overhead in locking/unlocking mutexes and waiting for other threads may even cause a net increase in runtimes. In addition, proposing and considering the moves may cause conflicts over shared resources, particularly if the image data cannot fit entirely into cache. Figure 3 can only be used to estimate the maximum possible speedup, actual improvements will fall short of this by an amount determined by the hardware and characteristics of the MCMC simulation to which speculative moves are applied.



**Figure 3. The number of steps required to perform 100 iterations using multiple processors. The serial implementation performs exactly one iteration in each step, the number of steps will always be 100 irrespective of  $p_r$ .**

## 6. 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 [14], tracking heads in a crowd or counting cells in slides of a tissue sample. We abstract this problem (for the sake of ease of understanding and testing) to recognising and counting artifacts (in this case circles) in an image.

In this case study an input image is first converted into a bitmap. After adding a small amount of random noise to improve stability when there are large regions of little or no natural variation, the bitmap is Sobel filtered to produce magnitude and orientation maps. These are then 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 addition of an artifact, the deletion of an artifact, 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 pro-

posed configuration matches the expected artifact properties, distribution of artifacts and amount of artifact overlap. The *likelihood* of the proposed configuration is obtained by comparing the proposed artifacts against the magnitude and orientation maps of the original 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 [5]. 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.

The artifact recognition program was implemented in C++, with the speculative moves mechanism implemented using pthreads. Threads were re-used, rather than created and destroyed on each iteration. The utilisation of these threads is performed on a best-effort basis as our program does not require complete synchronisation between all the threads at the start of each iteration. Instead a thread is used if and only if it is not already working on a previous iteration. This avoids the program waiting for a thread to finish processing a proposed move that will be certainly be rejected. Whilst some effort is made to cancel such move calculations, it is easier to just ignore the thread until the calculation completes normally. For the artifact recognition program it was found the two threaded program will use on average 1.99 threads in each iteration (threads are rarely required whilst calculating something else), while the four threaded version uses on average 3.6 threads.

For simplicity, we demonstrate our findings for test images. These are randomly generated white circles on a black background, with no other objects in the image. The circles were generated with the parameters (number, radii mean and variance) used by the prior calculations, with a check to avoid excessive overlapping of circles. A slight Gaussian blur was added to the image to make the circles easier to locate. More complex image processing examples have been studied, the reader is referred to [3, 13, 14, 16] as the applications per se are not the main focus of this paper.

Since the program execution time will vary due to the random nature of the MCMC method and the variation in input images, unless otherwise stated the execution times specified are averages over 20 runs of the program. Each run processed a different randomly generated image, using a different initial model and random number generator seeds. For each test a fixed number of iterations was performed (typically 10,000). Since the effective MCMC algorithm in use has not been modified, the same resultant models will be produced irrespective of how many threads/speculative moves are used. The traditional difficulty of determining when a MCMC program has ‘converged’ or completed its processing can therefore be ignored for the purposes of

judging the speculative move parallelisation method. Likewise the efficiency of the circle-finding algorithm and the fine tuning of its various parameters is not relevant beyond the program’s ability maintain a stable feature count throughout its execution.

## 7. Results

The following systems have been used for testing:

- AMD Athlon 64 X2 4400+ (dual-core), Linux 2.6.22-2
- Intel Xeon Dual-Processor, Linux 2.6.9-55
- Intel Pentium-D (dual core), Linux 2.6.18-36
- Intel Core2 Quad Q6600 (2x dual-core dies) Linux 2.6.18-36
- 56 Itanium2 processor SGI Altix, Linux 2.4-21-sgi306rp52<sup>1</sup> (up to 8 processors were used for the following tests).

For all the systems used in the tests, the addition of the threading mechanism and pthread locks increased sequential runtimes by less than 2%.

Figure 4 show a comparison of runtimes across all these systems for one set of tests (others tests carried out with different parameters provided similar results). In this case the move rejection rate was approximately 65%. Some systems made more efficient use of the speculative moves method than others (due to differing overheads) but in all cases the use of speculative moves reduced the runtime to between 45 and 80% of that of the single threaded implementation.

Next we consider the effects of varying the time taken to perform each iteration (obtained from the runtime of a program using only sequential execution). Two methods of varying the time-per-iteration can be used. The number of points sampled around each circle when performing likelihood calculations sample points can be increased so that the likelihood calculations for each circle take longer and involve more memory accesses. Alternatively the number of circles in the image can be increased, making the prior term take longer to process (most moves consider only the *change* they have on the likelihood, whereas the prior term must be recalculated in  $O(n^2)$  for each move).

Figure 5 shows the runtimes using one, two and four threads on the quad core Q6600. The per-iteration duration was varied by increasing the workload of the likelihood calculations whilst keeping the number of circles constant at 15, and the move rejection rate was fixed at 75%. For fast iterations the overhead involved in implementing speculative

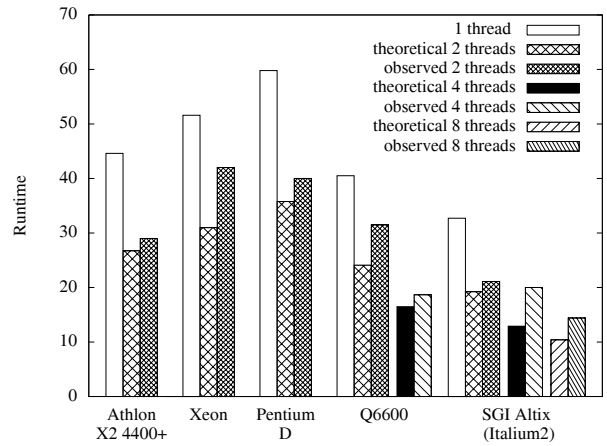


Figure 4. Speculative moves on different architectures,  $p_r = 0.65$

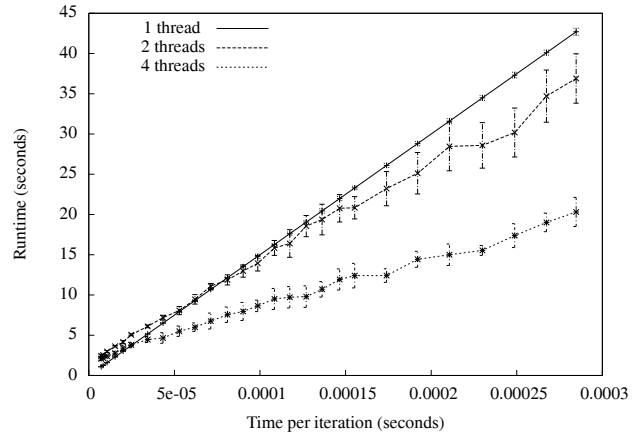


Figure 5. Runtime plotted against iteration time on the Q6600 (2x dual core),  $p_r = 0.75$

moves outweighs the benefits from the parallelisation. The points where the lines cross the 1-thread line represent how long each iteration must be before moves can be expected to start providing a real benefit (the point where the use of speculative moves ‘breaks even’, with the saving from considering moves simultaneously equalling the overhead required to implement that). These values are recorded for a number of alternative architectures in tables 1 and 2. As a point of reference, the circles program searching for 300 circles using a modest 32 sample points performed around 2000 iterations per second ( $500\mu s$  per iteration), whilst the vascular tree finding program from [3, 16] was generally performing 20 – 200 iterations per second ( $5 – 50ms$  an iteration). The tree crown finding program in [14] performed somewhere between ten to fifteen thousand iterations a second for small ( $200 \times 140$ ) images, processing larger images would be slower. We have found that many non-trivial MCMC applications will be well below the above iterations

<sup>1</sup>This computing facility was provided by the Centre for Scientific Computing of the University of Warwick with support from a Science Research Investment Fund grant

	Iteration Time ( $\mu s$ )	Iteration Rate ( $s^{-1}$ )
Xeon Dual-Processor	70	14 285
Pentium-D (dual core)	55	18 181
Q6600 (using 2 threads)	75	13 333
Q6600 (using 4 threads)	25	40 000

**Table 1. Breakeven point when  $p_r = 0.75$**

	Iteration Time ( $\mu s$ )	Iteration Rate ( $s^{-1}$ )
Xeon Dual-Processor	80	12 500
Pentium-D (dual core)	70	14 285
Q6600 (using 2 threads)	130	7 692
Q6600 (using 4 threads)	30	33 333

**Table 2. Breakeven point when  $p_r = 0.60$**

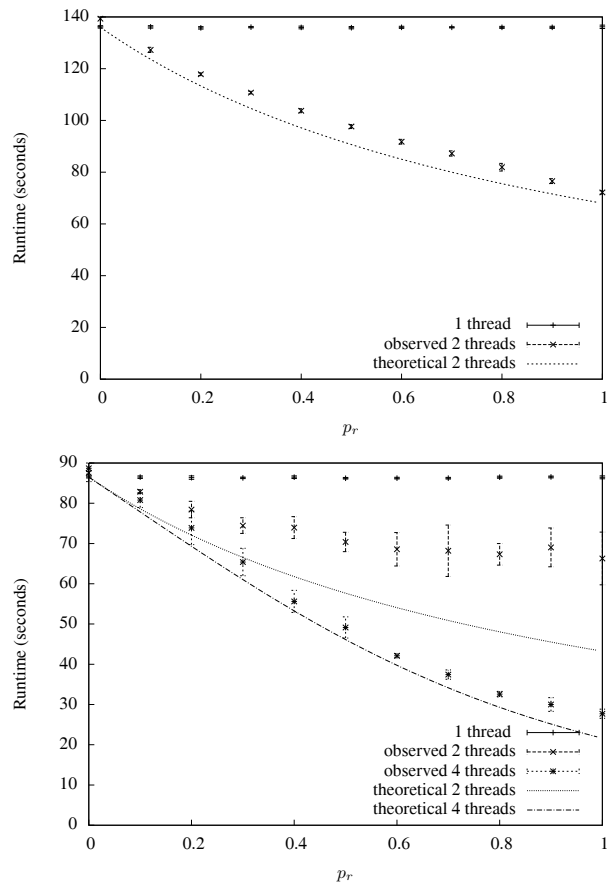
Machine	# threads		
	2	4	8
Xeon Dual-Processor	53	-	-
Pentium-D (dual core)	63	-	-
Athlon X2 (dual core)	75	-	-
Q6600 (2x dual core)	39	78	-
Altix (56 processor)	76	50	57

**Table 3. The percentage of the potential (theoretical) reduction in runtime that was achieved for a set of experiments where  $p_r = 0.65$ . Machines with higher values in the table are making more efficient use of their multiple processors.**

per second values and can therefore expect significant real-time savings by using speculative moves for real applications.

To determine the accuracy of predictions as the move rejection rate is varied, the program was modified to ignore the calculated Metropolis-Hastings ratio and accept or reject moves based on a presupplied probability (moves that added or removed features were disabled for this test, otherwise the runtimes would be dominated by the changes in the model size). The results for several machines are plotted in figure 7. The results for the Pentium D are a good match for the theoretical results given that the theoretical values assume ideal (and unachievable) conditions. The Q6600 results are more mixed, whilst using four threads yields results reasonably close to the theoretical bound, when using only two threads the results are substantially poorer.

This difference between architectures is further explored in table 3, where the percentage of the maximum runtime



**Figure 6. Runtime plotted against move rejection probability ( $p_r$ ) on the Pentium-D (top) and the Q6600 (bottom)**

reduction is displayed for the different architectures<sup>2</sup>. In the tests used to create this table  $p_r$  was  $\sim 0.78$ .

Considering all these results, the dual core machines (Pentium D and Athlon) gain more ( $\sim 10\%$ ) from speculative moves than the dual processor Xeon due to the increased overheads involved in communications between the Xeon's two processors. Compared to the Pentium-D the Athlon X2 achieves roughly 10% more of the potential out of speculative moves. This is due to the differences in Intel and AMD's dual core designs, whilst the Athlon X2's cores can communicate directly, inter-core communication on the Pentium-D must go through the motherboards chipset. For the Q6600 using only two threads (thus two cores) the breakeven point is comparable to the dual processor Xeon, yet when using all four cores the breakeven point and fulfilment of speculative move's potential was the best of those

<sup>2</sup>For example, consider a sequential program that takes 100s to run. If the theoretical maximum benefit from speculative moves would reduce that to 50s, yet experimental results showed the program ran in 75s, the percentage of the maximum runtime reduction that would be shown in the table would be 50.

machines examined ( $25\mu s$  per iteration and 78% respectively). The difference in results between using two and all four cores of the Q6600 is likely due to the Q6600's scheduler allocating the threads on to alternate dies (the Q6600 has two dual-core dies), a sensible strategy when each thread belongs to a different program but in this case counterproductive as frequent inter-core communication is required for speculative moves. In addition, when our MCMC program is using only two threads/cores, low priority processes would be scheduled on the two unutilised cores using up some of the shared non-processor resources that would otherwise have been used by the MCMC simulation (such as the inter-core communication channel). When all four cores were used such low priority processes were not getting as much processor time and so the inter-core communication channel was uncluttered, allowing the greater performance benefits from the speculative moves. Conversely, the Altix achieved most of the potential speedup when only two threads were used, but could only achieve 50-60% of the potential speedup when using more threads. The greatest reduction in runtime was achieved by the Altix using 8 threads (as in figure 4), but this was not done as efficiently as in other scenarios. The difference in efficiency for the Altix is due to the arrangement of its 56 Itanium 2 processors, communication between two processors is easier than communicating between 8.

## 8. Conclusion

Using our new 'speculative moves' approach takes advantage of multithreaded machines without altering any properties (other than runtime) of the Markov Chain. Our method can safely be used in conjunction with other parallelisation strategies, most notably Metropolis-Coupled Markov Chain Monte Carlo where each parallel chain can be processed on a separate machine, each being sped up using speculative moves. Speculative moves can be applied to any MCMC application performing fewer than some tens of thousands of iterations per second (dependant on machine architecture), many non-trivial and most long-running applications will fall into this category. When the average move rejection rate is high (i.e.  $> 50\%$ ) substantial savings in runtime can be expected. On typical MCMC applications, developers aim for rejection rates of around 75%, in which case runtimes could be reduced by up to 40% using a dual core machine, or up to 60% on a quad core machine. Multiprocessor machines will benefit, but CPUs with multiple processing cores on the same die will come closer to these limits. As multicore technology continues to improve the speculative move method will become more potent, applicable to an even wider range of applications and achieving greater performance improvements. By making the most of modern and future processor designs, specula-

tive moves will help make the use of MCMC more feasible in applications with time constraints (medical/security image processing) and existing MCMC application more productive, facilitating research in areas such as computational biology, physics, and econometrics.

## References

- [1] G. Altekar, S. Dwarkadas, J. P. Huelsenbeck, and F. Ronquist. Parallel metropolis-coupled markov chain monte carlo for bayesian phylogenetic inference. Technical Report 784, Department of Computer Science, University of Rochester, July 2002.
- [2] M. Bonamente, M. K. Joy, J. E. Carlstrom, E. D. Reese, and S. J. LaRoque. Markov chain monte carlo joint analysis of chandra xray imaging spectroscopy and sunyaevzel'dovich effect data. *The Astrophysical Journal*, 614(1):56–63, 2004.
- [3] D. C. K. Fan. *Bayesian Inference of Vascular Structure from Retinal Images*. PhD thesis, University of Warwick, May 2006.
- [4] X. Fenga, D. A. Buell, J. R. Rose, and P. J. Waddell. Parallel algorithms for bayesian phylogenetic inference. *Journal of Parallel and Distributed Computing*, 63:707–718, 2003.
- [5] P. Green. Reversible jump markov chain monte carlo computation and bayesian model determination, 1995.
- [6] P. J. Green. *Practical Markov Chain Monte Carlo*. Chapman and Hall, 1994.
- [7] P. J. Green. Mcmc in action: a tutorial. given at ISI, Helsinki, August 1999.
- [8] P. J. Green and A. Mira. Delayed rejection in reversible jump metropolis-hastings. *Biometrika*, 88(4):1035–1053, December 2001.
- [9] M. Harkness and P. Green. Parallel chains, delayed rejection and reversible jump mcmc for object recognition. In *British Machine Vision Conference*, 2000.
- [10] J. P. Huelsenbeck and F. Ronquist. Mrbayes: A program for the bayesian inference of phylogeny. Technical report, Department of Biology, University of Rochester, 2003.
- [11] M. Johannes and N. Polson. Mcmc methods for financial econometrics, 2002.
- [12] S. Li, D. K. Pearl, and H. Doss. Phylogenetic tree construction using markov chain monte carlo. *Journal of the American Statistical Association*, 1999.
- [13] C. C. McCulloch. *High Level Image Understanding via Bayesian Hierarchical Models*. PhD thesis, Institute of Statistics and Decision Sciences, Duke University, 1998.
- [14] 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*, volume 1, pages 661–4, 2005.
- [15] J. S. Rosenthal. Parallel computing and monte carlo algorithms, 1999.
- [16] E. Thonnes, A. Bhalerao, W. Kendall, and R. Wilson. A bayesian approach to inferring vascular tree structure from 2d imagery. In *International Conference on Image Processing*, volume 2, pages 937–940, 2002.
- [17] L. Tierney and A. Mira. Some adaptive monte carlo methods for bayesian inference. *Statistics in Medicine*, 18:2507–2515, 1999.