Accelerating Markov chain Monte Carlo via parallel predictive prefetching

HARVARD UNIVERSITY
Graduate School of Arts and Sciences

DISSERTATION ACCEPTANCE CERTIFICATE

The undersigned, appointed by the

School of Engineering and Applied Sciences

have examined a dissertation entitled:

"Accelerating Markov chain Monte Carlo via parallel predictive prefetching"

presented by : Elaine Lee Angelino

candidate for the degree of Doctor of Philosophy and here by certify that it is worthy of acceptance.

Signature [illegible]

Typed name: Professor M. Seltzer

Signature [illegible]

Typed name: Professor R. Adams

Signature Eddie Kohler

Typed name: Professor E. Kohler

Date: *August 11, 2014*

Accelerating Markov chain Monte Carlo via parallel predictive prefetching

A dissertation presented

by

Elaine Lee Angelino

to

The School of Engineering and Applied Sciences

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Computer Science

Harvard University

Cambridge, Massachusetts

August 2014

Dissertation Advisors Author

Professor Margo Seltzer and Professor Ryan P. Adams Elaine Lee Angelino

Accelerating Markov chain Monte Carlo via parallel predictive prefetching

Abstract

We present a general framework for accelerating a large class of widely used Markov chain Monte Carlo (MCMC) algorithms. This dissertation demonstrates that MCMC inference can be accelerated in a model of parallel computation that uses speculation to predict and complete computational work ahead of when it is known to be useful. By exploiting fast, iterative approximations to the target density, we can speculatively evaluate many potential future steps of the chain in parallel. In Bayesian inference problems, this approach can accelerate sampling from the target distribution, without compromising exactness, by exploiting subsets of data. It takes advantage of whatever parallel resources are available, but produces results exactly equivalent to standard serial execution. In the initial burn-in phase of chain evaluation, it achieves speedup over serial evaluation that is close to linear in the number of available cores.

Contents

Acknowledgements

This dissertation describes joint work with Eddie Kohler, Amos Waterland, Margo Seltzer, and Ryan P. Adams. I would like to thank Michael P. Brenner, Ekin Dogus Cubuk, Jonathan H. Huggins, Varun Kanade, Zhenming Liu, Dougal Maclaurin, and Robert Nishihara for helpful discussions, Alan Aspuru-Guzik, Johannes Hachmann and Roberto Olivares-Amaya for the use of the Clean Energy Project dataset and introduction to the cheminformatic feature set, and Michael Tingley for the derived features used here. Michael I. Jordan influenced writing that appears in Chapter 1. Kevin Swersky, David Duvenaud, Xinghao Pan and Joseph Gonzalez provided helpful comments on Chapter 6. Andrew C. Miller and Matthew J. Johnson provided many helpful comments on drafts of this manuscript. This work was partially funded by DARPA Young Faculty Award N66001-12-1-4219, the National Institutes of Health under Award Number 1R01LM010213-01, a Microsoft Research New Faculty Fellowship award, and Google.

I am deeply indebted to Margo Seltzer for supporting my various interests while steering me toward concrete and fulfilling directions. I would like to thank Ryan P. Adams for sharing his enthusiasm and ideas on diverse topics, and Eddie Kohler for incredible generosity with his time and systems expertise. I also thank Michael P. Brenner for putting up with me all these years. The Harvard School of Engineering and Applied Sciences has given me fine institutional support. I woud like to acknowledge Women in Machine Learning for having a profound impact on my research direction. Finally, I am very grateful to many individuals who have enhanced the last several years of my life; some of their names are listed below.

Margo Seltzer Ryan P. Adams Eddie Kohler Michael P. Brenner

Alecia McGregor Christina Cheuk

Roy Kishony Andrew Murray Pardis Sabeti Frank Solomon

Allison Craney

Adriana Gallegos

Dhruva Kothari Aaron Tjoa

Rianna Stefanakis

Bobby Thompson Ryan LaPerle Jaclyn Parks

Danny Goodman Marsha Berger Jonathan Goodman

Aviva Presser Aiden Erez Lieberman Aiden

Denny Kinlaw Ben Belknap

Daniel Yamins Janice Yamins David Yamins

Justin T. Riley Eric Jones Doug Fritz

Jason Rosenfeld

Uri Braun David Holland Daniel Margo Peter Macko Nicholas Murphy

Kiran-Kumar Muniswamy-Reddy Robin Smogor Susan Welby

Marc Chiarini Lisa Lowy

Ann Marie King Marie Dooley Lisa Frazier Meg Hastings Julie Holbrook

Peter Bailis Karthik Dantu Brian Kate Jason Waterman Jim Waldo

Philip Guo Nils Napp Kirstin Petersen

Heather Pon-Barry Alice Gao Sophia Shao Stacy Wong Elif Yamangil

Neena Kamath Amy Tai Joy Zhang Raina Masand

Shelby Lin Maddie Boyd Julie Monrad Gabrielle Ehrlich

Jeremy Rassen   Pete Wahl   Sebastian Schneeweiss

Michael Mitzenmacher   Naveen Sinha   Brent Heeringa   Michael Goodrich

Jonathan Appavoo   Efthimios Kaxiras

Steve Chong   Greg Morrisett   Radhika Nagpal   Barbara Grosz

Jeremy Gunawardena

Christos Papadimitriou

Leslie Valiant

Elena Agapie   Anna Huang   Varun Kanade   Justin Thaler

Amos Waterland   Dogus Cubuk   Miguel Aljacen   Miriam Huntley   Ben Good

James Zou   Zhenming Liu   Jon Ullman   Thomas Steinke   Mark Bun

Kenneth Arnold   Pao Siangliulue   Bo Waggoner   Sam Wiseman

Margo Levine   Marie Dahleh   Beth Chen   Sarah Kostinski

Michael S. Kester   Jemila C. Kester   Sarah M. Kester   Hannah J. Kester

Gregory Valiant   Sham Kakade

Teagan Seltzer   Tynan Seltzer   Keith Bostic

Mary Baker

Eyal Dechter   Finale Doshi-Velez   David Duvenaud   Yakir Reshef

Diana Cai   Michael Gelbart   Scott Linderman   Dougal Maclaurin   Robert Nishihara

Oren Rippel   Jasper Snoek   Kevin Swersky   Brian Zhang

Matt Johnson   Jonathan Huggins

Andy Miller

Zorana Zeravcic   Andrej Mesaros

Norman J. Angelino   Theresa Yi-Yuan Lee   Kenneth Lee Angelino   Keith Lee Angelino

To my parents

Chapter 1

Motivation and summary

A central tool of modern data analysis is *inference*, the process of estimating structure in data via probabilistic modeling. The goal is to recover the parameters of a probabilistic description of data, given a set of observations. In particular, Bayesian inference uses Bayes' rule to update a probabilistic description of model parameters as more data are observed. Sadly, inference is computationally expensive when the underlying functions are high-dimensional and/or full of many local optima, as is typical with large datasets. In general, there are no analytic solutions to these problems; there are approximate and simulated approaches, but these are often slow and do not naturally leverage modern computing resources, such as clouds.

Inference is dominated by two approaches: using *optimization* procedures to find the best model parameter setting and the Bayesian approach of *integrating* with respect to the relative probabilities of various parameter settings. This thesis focuses on Bayesian procedures, which have been mostly absent in discussions of large-scale inference until very recently. While there have been recent successes in scaling inference procedures, most have focused on optimization.

The main computation in Bayesian inference is that of the *posterior* density $\pi(\theta \,|\, \mathbf{x})$ over the parameters θ to a probabilistic model, given a set of observed data $\mathbf{x} = \{x_1, \dots, x_n\}$. The posterior is proportional to the product of two other probability densities, a *likelihood* $\pi(\mathbf{x} \mid \theta)$ describing the probability of the data, given the model, and a *prior* $\pi_0(\theta)$ over the model parameters. Bayesian inference is appealing because the posterior density encodes uncertainty over model parameters; this uncertainty can then be propagated to downstream applications. However, there are often no analytic solutions to useful functions of the posterior, such as expectations; typically these involve integrating over the parameters. While samples from the posterior can be used to estimate quantities of interest, there is usually no analytic way to obtain them. This motivates approximate sampling-based methods such as *Markov chain Monte Carlo* (MCMC) and *importance sampling.* Unfortunately, these methods are difficult to scale, which has inhibited their application to large datasets.

This thesis focuses on MCMC, a widely used, powerful and general technique for both optimization and Bayesian inference. In the optimization setting, it stochastically searches a parameter space for the best setting of θ. In Bayesian inference, it produces a sequence of samples drawn from a sequence of distributions that converge to the posterior distribution. These algorithms are typically slow to converge so they must be run for many iterations before they yield useful output. Furthermore, they are inherently *serial* and thus, in general, do not parallelize well.

Reliance on serial algorithms is a great frustration given the power of today's scientific computing environments, which are highly parallel. Researchers have routine access to hundreds to thousands of parallel cores in multicore environments, where computational work can be distributed over multiple cores that are able to communicate with one another. Thus, our ability to perform large-scale Bayesian inference is limited by our algorithms, not our computational resources.

The pseudocode in Algorithm 1 illustrates the serial nature of many Bayesian inference procedures: start with some initial setting of model parameters θ_0, then iteratively select the next parameter setting θ_1 from some set of choices that depend on θ_0, then θ_2 from choices that depend on θ_1, and so on. Each iteration can take a long time – *e.g.*, because selecting θ_t

Algorithm 1 Serial Bayesian inference

```
Specify a dataset $\mathbf{x}$, a posterior density $\pi(\theta \mid \mathbf{x})$ and an initial parameter setting $\theta_0$.
for $t$ in $0, \ldots, T$ do
    Generate one or more parameter settings $\{\theta'\}$ that depend on $\theta_t$.
    Select $\theta_{t+1}$ from $\{\theta'\}$ by comparing the evaluations of $\{\pi(\theta' \mid \mathbf{x})\}$ to $\pi(\theta \mid \mathbf{x})$.
end for
Output some function of $\theta_1, \theta_2, \theta_3, \ldots$.
```

for $t > 0$ depends on the computationally expensive evaluation of $\pi(\theta_t \mid \mathbf{x})$. If we had N cores and could perform N iterations at a time in parallel, then we could speed-up execution by a factor of N. However, since each iteration depends on the last, it is not possible to skip ahead to later iterations without first completing earlier ones. Specifically, the iteration indexed by t produces θ_{t+1} in a way that depends on knowing θ_t, which in turn depends on $\theta_{t-1}, \theta_{t-2}, \ldots, \theta_0$, only the last of which is known initially.

That said, there is nothing to stop us from materializing predictions for θ^t and executing the corresponding iterations on parallel cores. This is a form of *speculative execution*, the technique of optimistically performing computational work that might be eventually useful. **This dissertation demonstrates that MCMC inference can be accelerated in a model of parallel computation that uses speculation to predict and complete computational work ahead of when it is known to be useful.**

Below, we outline how the remaining chapters demonstrate the veracity of this thesis statement. In Chapter 2, we review Markov chain Monte Carlo, an algorithmic approach for stochastically estimating the expectation of a function with respect to a probability distribution. Computing such an expectation might be an intractable task, *e.g.*, its exact calculation might involve a sum of exponentially many values or an integral with no known analytic solution. MCMC combines two powerful ideas – Markov chains and Monte Carlo integration – and we begin by explaining the basic theory and properties of all three. In particular, the serial nature and convergence behavior of MCMC algorithms derive from their underlying use of Markov chains. The Metropolis–Hastings (MH) algorithm provides a concrete introduction to MCMC; it is a simple and canonical algorithm that illustrates the challenges and limitations of MCMC. The rest of the chapter categorizes existing MCMC algorithms according to their strategies for improving on naïve algorithms such as MH. The algorithms in the first of two broad categories attempt to decrease the time to reach convergence; those in the second make use of parallel resources. We do not provide a complete review of all MCMC algorithms, which have been reviewed elsewhere, but we do thoroughly summarize existing parallel MCMC algorithms that use speculative techniques, called *prefetching* in this literature. Finally because this thesis is motivated by large-scale Bayesian inference, the chapter ends with a summary of MCMC algorithms recently proposed for this setting.

The core intellectual contributions of this thesis are in Chapter 3, where we propose and analyze a new class of prefetching MCMC algorithms. First, we provide a mathematical language for describing a large class of MCMC algorithms that can be mapped to, and would benefit from, prefetching. This treatment is more formal and general than what has been provided by prior prefetching literature but is designed to motivate prefetching and elucidate its feasibility and validity. For concreteness, the remainder of the thesis focuses on Metropolis–Hastings, where prefetching requires speculating about the outcome of a binary condition at each iteration of the algorithm. This motivates *predictive prefetching*, a principled framework for exploiting predictions about these binary outcomes so as to most effectively allocate parallel resources. The goal is to maximize the expected speed-up relative to serial execution, given parallel cores and predictive information. We derive predictors for the setting of large-scale Bayesian inference that we later use directly in the empirical studies of Chapter 5. Finally, since perfect predictions are not normally available, we analyze the performance of predictive prefetching in terms of expected speed-up as a function of predictor accuracy and the number of parallel cores.

Chapter 4 describes the design and implementation of a practical parallel system for predictive prefetching. The system architecture follows a master-worker pattern in which a single master core maintains information about computational work that might be useful, determines what work is carried out by the remaining worker cores, and records the results of these computations. The master maintains data structures that organize the results of potentially useful increments of computational work, plus related information. These increments of work include all those that exactly correspond to equivalent serial execution and are eventually identified as such with absolute certainty. Workers request work from the master whenever they are available, the master replies to each worker with a specification of the work to do, and workers send computed results back to the master. The system guarantees results equal to serial execution, *i.e.*, invariant to the number of cores used. Since MCMC algorithms are stochastic, this guarantee depends critically on correct management of the source of (pseudo)randomness. This issue is subtle and the solution presented here is more careful than any provided in prior literature on prefetching. The implementation includes a simple plug-in interface for specifying a concrete instantiation of a MH algorithm via user-defined functions. We also provide remaining details about specific implementation choices and artifacts.

Next, in Chapter 5, we present an empirical evaluation of the parallel implementation of predictive prefetching in a real research computing environment. We select and implement concrete large-scale Bayesian inference problems involving both synthetic and real datasets. The efficiency of predictive prefetching depends on the behavior of MH, which in turn depends in a sensitive fashion on parameters that are typically hand-tuned by practitioners according to heuristic guidelines. Furthermore, this behavior changes – often dramatically – over the course of running a single instantiation of the algorithm. To execute reasonably calibrated experiments, we identify an adaptive MH scheme that eliminates this tuning problem and requires only a simple extension to our original implementation for MH. We clearly describe a framework for assessing chain convergence, which we use to identify different regimes of chain behavior. We present and discuss empirical results for speed-up as a function of the number of parallel cores used, measured relative to a baseline system implementation with one master and one worker. The chapter ends with a discussion of the overheads of our system.

Finally, in Chapter 6 we distill the conclusions of this thesis, including lessons learned and a map of possible extensions to this work. We will have demonstrated effective use of relatively naïve prediction strategies, therefore we identify additional promising strategies for predictive prefetching, emphasizing generic methods based on constructing approximations to a target density. We also outline technical challenges for predictive prefetching in the context of more sophisticated MCMC algorithms, then propose and justify potential solutions. We end with a broad discussion of opportunities for applying speculative execution to algorithms ranging across various properties: stochastic versus deterministic, exact versus approximate or heuristic, discrete versus continuous.

Chapter 2

Markov chain Monte Carlo

Markov chain Monte Carlo (MCMC) is a widely used, powerful technique for estimating statistics of an arbitrary distribution Π defined over a state space $\mathcal{X}$. MCMC simulates a random walk that produces a sequence of samples drawn from a sequence of distributions that converges to Π. MCMC is typically employed when samples from, or statistics of, a distribution cannot be obtained analytically, as is often the case with complex, high-dimensional systems arising across disciplines, *e.g.*, estimating bulk material properties from molecular dynamics physics simulations or inferring the parameters of Bayesian probabilistic models describing large datasets.

In this chapter, we first review the two powerful tools underlying MCMC algorithms – *Markov chains* and *Monte Carlo* methods. Next, we introduce MCMC via the well-known *Metropolis-Hastings* algorithm, both as a way to concretely exemplify relevant concepts and to motivate a large body of research whose goal is to design more efficient MCMC algorithms. We then provide an overview of different classes of these approaches, with greater focus on the areas that together provide the foundation for a new approach to large-scale MCMC that we present in the next chapter.

2.1 Markov chains

Let $\mathcal{X}$ be a discrete or continuous state space and let $x, x' \in \mathcal{X}$ denote states. A Markov chain is a discrete-time stochastic process governed by a transition operator $T(x \to x')$ that specifies the probability of transitioning from a current state x to some next state x'. It is *memoryless* in the sense that its future behavior depends only on the current state and is independent of its past history – this is known as the Markov property.

Many systems can be modeled by Markov chains. For example, an unbiased random walk on a one-dimensional lattice is described by a Markov chain. The integers modulo k can be used to index a finite lattice of k states, in which case $\mathcal{X} = \mathbb{Z}_k$. The transition operator,

$$T(x \to x - 1 \mod k) = T(x \to x + 1 \mod k) = T(x \to x \mod k) = \frac{1}{3}, \tag{2.1}$$

describes a random walk on the lattice, with periodic boundaries, that at each time step either moves to the 'left' or 'right' by one unit, or stays put, where the three scenarios are equiprobable. Here, the stationary distribution is simply the uniform distribution over $\mathbb{Z}_k$.

Given an initial distribution $P^0(x)$ over $\mathcal{X}$, a Markov chain evolves this distribution from one time point to the next through iterative application of the transition operator; after t steps let us call this distribution $P^t(x)$. Direct *simulation* of a Markov chain follows this iterative construction and leads to inherently serial implementations. We are interested in Markov chains that *converge* to a unique *stationary distribution* $\pi(x)$ in the sense that

$$\lim_{t \to \infty} P^t(x) \to \pi(x),$$

for any initial distribution $P^0(x)$.

The *speed of convergence* or *mixing time* of a Markov chain measures how quickly $P^t(x)$ approaches $\pi(x)$; it is typically defined with respect to a distance measure between probability distributions and a threshold. For example, it could be defined as the minimum or expected number of steps t such that $D_{\mathrm{KL}}(\pi(x) \parallel P^t(x)) < \epsilon$, for some appropriate $\epsilon > 0$, where $D_{\mathrm{KL}}(P \| Q)$ is the *Kullback-Leibler divergence* of two distributions P and Q, and we think of Q as an approximation to P (Kullback and Leibler, 1951). Convergence behavior depends on the properties of the state space $\mathcal{X}$ – *e.g.*, whether it is discrete or continuous, its dimensionality – and the behavior of the transition operator.

For example, consider a simulation of a one-dimensional, k-state random walk, described by the transition operator in Eq. 2.1. The mixing time is $O(k^2)$, *i.e.*, the simulation requires $O(k^2)$ steps to 'forget' the initial condition and look reasonably like the uniform stationary distribution. In contrast, consider a deterministic transition operator that always moves to the 'right', *i.e.*, $T(x \to x+1) = 1$. This time, simulation requires only $O(k)$ steps to approach the uniform stationary distribution. While this simple example represents an extreme case that is not useful for typical applications, it illustrates how two Markov chains can have the same stationary distribution but different convergence behavior. A major area of Markov chain research is understanding how to design efficient transition operators that converge quickly, as doing so has direct practical consequences for their simulation.

For a transition operator $T(x \to x')$ to have $\pi(x)$ as its stationary distribution, its application must leave $\pi(x)$ invariant over the entire space, *i.e.*,

$$\sum_{x \in \mathcal{X}} T(x \to x')\pi(x) = \pi(x'), \quad \forall x' \in \mathcal{X}$$

for a discrete state space, or

$$\int_{\mathcal{X}} T(x \to x')\pi(x)dx = \pi(x'), \quad \forall x' \in \mathcal{X} \tag{2.2}$$

for a continuous state space; this thesis will focus on continuous state spaces. For the stationary distribution to be unique, *i.e.*, not depend on the initial distribution, the Markov chain must be *irreducible*: for any $x, x' \in \mathcal{X}$ such that $\pi(x), \pi(x') > 0$, it must be possible to reach x' from x in a finite number of steps. A powerful application of Markov chains involves designing a transition operator that has as its stationary distribution some target distribution of interest – this is the main idea behind Markov chain Monte Carlo methods.

In restricted cases it is easy to show that a transition operator has a certain stationary distribution. Notably, when a transition operator $T(x \to x')$ is *reversible*, it satisfies *detailed balance* with respect to a distribution $\pi(x)$,

$$T(x \to x')\pi(x) = T(x' \to x)\pi(x'), \tag{2.3}$$

and it is easy to show that $\pi(x)$ is its stationary distribution. Integrating over $\mathcal{X}$ on both sides gives:

$$\begin{aligned}\int_{\mathcal{X}} T(x \to x')\pi(x)dx &= \int_{\mathcal{X}} T(x' \to x)\pi(x')dx \\ &= \pi(x') \int_{\mathcal{X}} T(x' \to x)dx \\ &= \pi(x'),\end{aligned}$$

which is precisely the required condition from Eq. 2.2. We can interpret Eq. 2.3 as stating that, for a reversible Markov chain starting from its stationary distribution, any transition $x \to x'$ is equilibrated by the corresponding reverse transition $x' \to x$. As we will see, many MCMC methods are based on deriving reversible transition operators. A transition operator that is not reversible is called *non-reversible*; it is generally more difficult to manipulate and prove statements about these.

For a formal introduction to Markov chains, see the book by Meyn and Tweedie (1993).

2.2 Monte Carlo methods

Monte Carlo methods are a broad class of algorithms that simulate many repeated random samples to estimate some quantity of interest. For example, the following procedure is a form of *Monte Carlo integration* that estimates the area under any positive function

$f : [a, b] \to \mathbb{R}^+$, where $-\infty < a < b < \infty$:

1. Draw a box around f with vertical boundaries set by the interval $[a, b]$ and horizontal boundaries set by 0 and an upper bound m on the maximum value of f in the interval.
2. Sample a large number of random points (x, y) uniformly within the box and for each, determine whether the point falls below or above f by computing whether $f(x) < y$.
3. Let r be the fraction of points such that $f(x) < y$. Since the total area of the box is $m(b - a)$, multiplying by r provides an estimate for $\int_a^b f(x)dx$.

More generally, when we can think of an integral as an expectation, Monte Carlo integration invokes the law of large numbers to estimate this expectation via a sample average. Specifically, if we can write an integral as the expectation of a function $f(x)$ with respect to a distribution Π with probability density function $\pi(x)$,

$$\mathrm{E}_\Pi(f) = \int f(x)\pi(x)dx, \tag{2.4}$$

then we can estimate this integral by averaging over a set of samples $\{x_n\}_{n=1}^N$ from Π as:

$$\bar{f}_N \equiv \frac{1}{N}\sum_{n=1}^{N} f(x_n).$$

Since the samples are independent, as long as the expectation in Eq. 2.4 exists and is finite, this sum obeys the law of large numbers. Hence, the estimate is unbiased and its variance scales as the inverse sample size $1/N$, or equivalently, its error scales as $1/\sqrt{N}$. In our example above, the integral of $f(x)$ on the interval $[a, b]$ can be thought of as an expectation with respect to the uniform distribution on $[a, b]$.

Monte Carlo integration thus requires sampling from a distribution, which is sometimes straightforward, as with the uniform and normal distributions, but in general requires numerical simulation. Below, we describe two additional Monte Carlo methods that address this issue in restricted settings: *rejection sampling* and *importance sampling*. Their limitations and inefficiencies will help motivate Markov chain Monte Carlo methods, which are more sophisticated but related techniques. For simplicity, we describe these procedures with respect to one-dimensional normalized probability densities; both can be generalized.

2.2.1 Rejection sampling

Rejection sampling uses one distribution to sample from another by exploiting information relating the two; von Neumann (1951) provided an algorithm for this method. Suppose that we want to sample from a *target distribution* Π with probability density function $\pi(x)$. Suppose further that we can sample from a *proposal distribution* Q whose probability density function $q(x)$ can be scaled by a constant factor γ to provide an upper bound on $\pi(x)$, *e.g.*, we might be able to scale a normal distribution so that our distribution of interest lies below it everywhere. If we satisfy these requirements, then we can use rejection sampling to generate *proposals* from Q that we stochastically *accept* or *reject* according to the relative difference between $\gamma q(x)$ and $\pi(x)$. Specifically, to produce one sample:

1. Generate a proposal x by drawing a sample from the proposal distribution Q.

2. Draw a sample y uniformly from the interval $[0, \gamma q(x)]$.

3. If $y < \pi(x)$, accept x. Otherwise, reject x and return to Step 1.

Rejection sampling is most efficient in the limit where the scaled proposal density equals the target density, in which case all proposals are accepted. More generally, in expectation, this procedure accepts proposals at a rate given by $\int \pi(x)/(\gamma q(x))dx \leq 1$.

2.2.2 Importance sampling

Similar to rejection sampling, importance sampling also uses information from one distribution to sample from another, but with fewer restrictions. Suppose we have distributions Π

and Q as above, where this time we can simply think of $q(x)$ as an approximation to $\pi(x)$; *i.e.*, we do not require some $\gamma q(x)$ that is an upper bound to $\pi(x)$. Suppose we want to compute the expectation of some function $f(x)$ with respect to the distribution Π:

$$\mathrm{E}_{\Pi}(f(x)) = \int f(x)\pi(x)dx.$$

By multiplying and dividing by $q(x)$ inside the integral,

$$\mathrm{E}_{\Pi}(f(x)) = \int \frac{f(x)\pi(x)}{q(x)} q(x)dx \equiv \mathrm{E}_{Q}\left(f(x)w(x)\right),$$

we change nothing, but can interpret this new expression as the expectation of $f(x)$ weighted by $w(x) = p(x)/q(x)$ with respect to Q. We can Monte Carlo estimate this integral using a set of samples $\{x_n\}_{n=1}^{N}$ from Q:

$$\frac{1}{N}\sum_{n=1}^{N} f(x_n)w(x_n).$$

The quality of this estimator depends on how much $f(x)w(x)$ varies – ideally this quantity would be constant with respect to x. Some historical notes and a list of references on importance sampling can be found in the textbook by Gelman et al. (2003).

2.2.3 Limitations of Monte Carlo sampling

The primary limitation of both rejection sampling and importance sampling is that for these methods to be feasible and practical, each requires a proposal distribution that can be sampled easily and is in some sense close to the target distribution. To produce samples, both methods use a set of independent samples from the proposal distribution; rejection sampling selects from among the proposals and importance sampling 'fixes up' the proposals by assigning each a weight.

2.3 Markov chain Monte Carlo

Markov chain Monte Carlo (MCMC) methods simulate a Markov chain whose stationary distribution is equal to a target distribution of interest. When this Markov chain is simulated, it produces samples from a sequence of distributions that asymptotically equals the target distribution. The principles of Monte Carlo integration, estimation and sampling thus apply to these samples in the asymptotic limit. Concretely, for a Markov chain started from its stationary distribution Π with density $\pi(x)$, a sequence of N samples $\{x_n\}_{n=1}^N$ can be used to estimate an expectation $E_\Pi(f) = \int_X f(x)\pi(x)dx$ using Monte Carlo integration via the sample average $\bar{f}_N = \frac{1}{N}\sum_{n=1}^N f(x_n)$. The efficiency of a MCMC transition operator can be analyzed with respect to both the variance of this estimator, also known as the *asymptotic variance*, as well as the speed of convergence or mixing time. In practice, we use samples produced by simulated chains of finite length, typically started away from stationarity. The materialized sequence of samples obeys the Markov property and is correlated, which is in contrast to the independent samples obtained by simple Monte Carlo methods such as rejection sampling and importance sampling.

The remaining sections of this chapter give an incomplete overview of MCMC algorithms for sampling applications, with greater emphasis on certain procedures either for the purpose of providing general background or to review those most directly related to this thesis. First, we describe the Metropolis-Hastings (MH) algorithm, a canonical and simple MCMC method. We use MH to build some intuition for the behavior of MCMC algorithms, and to illustrate its limitations that motivate more sophisticated approaches. The following two sections classify these further approaches into serial algorithms designed to converge more quickly than MH and parallel algorithms. Finally, we briefly review MCMC algorithms that exploit common features of Bayesian inference problems. For a general introduction to MCMC, see the highly motivating review by Diaconis (2008).

2.4 Metropolis-Hastings (MH)

The Metropolis-Hastings (MH) algorithm simulates a Markov chain, over a state space $\mathcal{X}$, with stationary distribution equal to some target distribution of interest. Given an initial state x_0, a target distribution π and a proposal function $q(x'|x)$, MH generates a sequence of states $x_1, \ldots, x_T \in \mathcal{X}$ drawn from a sequence of distributions that converges to the target.[1] We provide pseudocode for MH in Algorithm 2. Each iteration, a proposal for the next state x' is drawn from the proposal distribution, conditioned on the current state x; *e.g.*, a common choice is to sample from a Gaussian centered at x. The proposal is stochastically accepted with probability given by the *acceptance ratio*,

$$r = \frac{\pi(x')q(x \mid x')}{\pi(x)q(x' \mid x)}, \tag{2.5}$$

via comparison to a random variate u drawn uniformly from the interval $[0, 1]$. If $u < r$, then the next state is set to the proposal, otherwise, the proposal is rejected and the next state is set to the current state. MH is a generalization of the *Metropolis algorithm* (Metropolis et al., 1953), which requires the proposal distribution to be symmetric, *i.e.*, $q(x'|x) = q(x|x')$, in which case the acceptance ratio is simply $r = \pi(x')/\pi(x)$. Hastings (1970) later relaxed this by showing that the proposal distribution could be arbitrary.

The MH algorithm can be viewed as a biased random walk that always accepts proposals when $\pi(x')q(x \mid x') > \pi(x)q(x' \mid x)$ and stochastically rejects them otherwise; for a symmetric proposal distribution, these scenarios can be interpreted as accepting 'uphill' proposals and stochastically rejecting 'downhill' proposals. We can see that the stationary distribution is indeed π by showing that the MH transition operator satisfies detailed balance, as defined

[1] As is common in the literature, we will henceforth use the same symbol to refer to both a distribution and its probability density function; the interpretation should be clear from context.

Algorithm 2 Metropolis-Hastings

```
**Input:** Initial state $x_0$, number of iterations $T$, target $\pi(x)$, proposal $q(x' \mid x)$
**Output:** Samples $x_1, \ldots, x_T$
for $t$ in $0, \ldots, T-1$ do
    $x' \sim q(x' \mid x_t)$                                        ▷ Generate proposal
    $r \leftarrow \frac{\pi(x')q(x_t \mid x')}{\pi(x_t)q(x' \mid x_t)}$        ▷ Compute acceptance ratio
    $u \sim \mathrm{Unif}(0,1)$                                     ▷ Draw random number
    if $u < r$ then
        $x_{t+1} \leftarrow x'$                                     ▷ Accept proposal
    else
        $x_{t+1} \leftarrow x_t$                                    ▷ Reject proposal
    end if
end for
```

in Eq. 2.3. From the algorithm description, the MH transition operator is:

$$T(x \to x') = \min(1, r) q(x' \mid x) = \min\left(1, \frac{\pi(x')q(x \mid x')}{\pi(x)q(x' \mid x)}\right) q(x' \mid x).$$

We can verify the detailed balance condition as follows:

$$\begin{aligned} T(x \to x')\pi(x) &= \min\left(1, \frac{\pi(x')q(x \mid x')}{\pi(x)q(x' \mid x)}\right) q(x' \mid x)\pi(x) \\ &= \min\left(\pi(x)q(x' \mid x), \pi(x')q(x \mid x')\right) \\ &= \min\left(\frac{\pi(x)q(x' \mid x)}{\pi(x')q(x \mid x')}, 1\right) q(x \mid x')\pi(x') \\ &= T(x' \to x)\pi(x'). \end{aligned}$$

2.4.1 Factors affecting the behavior of MH

The MH algorithm is both simple to implement and quite general; it is thus appealing and widely applicable. However, the MH algorithm has a major drawback – it can be slow to converge. This is due to the fact that the steps of the underlying Markov chain are correlated, which can be viewed as random walk or diffusive behavior. One broad strategy for increasing the efficiency of MCMC methods is to design transition operators that behave less like simple diffusion; we survey several techniques for doing so in the next section.

Within the MH framework and given a target density, the variable parameters are the proposal distribution and the initial condition. Let us first consider the proposal distribution. For example, for a one-dimensional continuous target density, if we restrict the proposal distribution to be Gaussian and centered at the current state, $q(x' \mid x) = \mathcal{N}(x' \mid x, \sigma^2)$, then there is a single tuning parameter: the distribution's standard deviation σ, which gives the expected 'step size' of the proposal with respect to the current state. This affects the MH *acceptance rate*, which we also refer to as the *acceptance probability*, *i.e.*, the fraction of proposals that are accepted.

To illustrate the relationship between the proposal distribution and the acceptance rate, consider unimodal target and proposal distributions. Suppose that we are able to initialize MH at a state close to the target distribution's mode with respect to its width. Intuitively, if the proposal step size is large compared to the width of the target, then proposals will tend to fall in faraway, low-probability regions, resulting in a low acceptance rate. On the other hand, if the step size is very small, then the target density at the proposal will be very close to that at the current state, in which case the algorithm will tend to accept proposals, but the samples will be highly correlated and the chain will take a long time explore the area under the target density. This suggests that there is some notion of an optimal MH acceptance rate corresponding to some intermediate proposal step size.

A classic result is that the optimal value for the MH acceptance rate is 0.234, derived for the scenario where the target and proposal distributions are multidimensional Gaussians, in the limits where the chain has converged and the number of dimensions tends to infinity (Roberts et al., 1997). A heuristic widely followed by practitioners is to tune the proposal distribution to obtain an observed acceptance rate of about 0.234.

The sensitivity of the acceptance rate as a function of the proposal distribution also explains why the MH algorithm has trouble sampling from multimodal target densities. When modes are far apart compared to the widths of the peaks around them, they are separated by low-probability regions that are difficult for a simulated MH chain to traverse. In these cases, the MH algorithm tends to get 'stuck' for many iterations around local modes, instead of sampling globally from the entire distribution. In practice, a MH simulation tends to find the mode closest to the initial state and then samples the area around this mode.

Given target and proposal distributions, the only other specification required by the MH algorithm is an initial state. Clearly, it is desirable for the initial state to be close to some probable region of the target density – a 'bad' initial state combined with the random walk nature of chain simulation yields initial samples that are not representative of the target. This initial portion of a MCMC simulation, before convergence, is sometimes called *burn-in*.

The behavior of a MCMC simulation during burn-in is different from that after convergence, because the shape of the target density differs far from versus close to the bulk of its mass. Specifically, the target density tends to be 'flatter' or 'less steep' around a mode compared to less probable regions. This characterization interacts with proposal generation, resulting in acceptance behavior that changes from burn-in to convergence.

To illustrate differences in MCMC behavior between burn-in and convergence, consider MH for a Gaussian target distribution. Typically, a MCMC simulation is initiated at some informed guess that is still somewhat far from higher probability regions of the target; assuming it is well-behaved, the chain should eventually spend more time in these regions. A Gaussian distribution has its mass concentrated around a single mode. A region close to this mode can be well-approximated by an upside down parabola – a quadratic function – while the tails fall off exponentially quickly. Suppose also that our proposal distribution is symmetric and its width is not large compared to the width of the target. In the region close to the target mode, the target densities evaluated at two nearby states will tend to be comparable values. In the context of MH, the acceptance ratio r will be well within the interval $[0, 1]$ and the decision to accept or reject a proposal depends on both r and the random variate u. If we consider two nearby states in the tail regions, then the target density evaluated at one will be exponentially smaller than the other. Here, the acceptance ratio r

will be close to either 0 or 1, so the random variate u has little influence over whether a proposal is accepted. As we will see later, these differences between chains during burn-in and convergence have implications for the performance of our new approach to MCMC as well as our empirical studies.

2.5 MCMC methods for faster convergence

In this section, we survey classes of MCMC algorithms designed to converge more quickly than the MH algorithm by reducing the correlation between successive states. We do not provide a thorough review, as the methods we develop in this thesis do not build directly on these techniques. However, we do describe specific algorithms both for concreteness, and because we will later consider them within the context of *predictive prefetching*, a new framework we present in Chapter 3.

2.5.1 Auxiliary variable methods

Given a target density $\pi(x)$, we can introduce an *auxiliary variable* y and define a new density $\pi(x, y)$ such that $\int \pi(x, y)dy = \pi(x)$, *i.e.*, marginalizing out y the yields the target. Auxiliary variable methods design MCMC sampling schemes over the space of a new joint distribution; after sampling from $\pi(x, y)$, one obtains samples from $\pi(x)$ simply by ignoring the y values. While it would seem less desirable to sample from a higher dimensional space, it is possible to design transition operators over the joint space that marginally sample from the target in a way that is more efficient than Metropolis-Hastings.

For example, consider a one-dimensional target density $\pi(x) : \mathbb{R} \to \mathbb{R}^+$. Sampling from $\pi(x)$ yields a sequence of samples along the real line. Now consider a representation of the target in the (x, y)-plane such that $y = \pi(x)$. If we sample a set of points $\{(x_i, y_i)\}$ uniformly within the two-dimensional area between $\pi(x)$ and the x-axis, then marginally, the $\{x_i\}$ are samples from $\pi(x)$. Below, we summarize two auxiliary variable methods: slice sampling and Hamiltonian Monte Carlo.

Slice sampling methods are based on the above idea, sampling from the joint distribution $\pi(x, y)$ by iteratively sampling each variable marginally (Neal, 2003). Given some state x_i, the procedure constructs y_i and then the next x_{i+1} as follows:

1. Sample $y_i \sim \pi(y_i \,|\, x_i)$ by sampling uniformly from the (vertical) interval $[0, \pi(x_i)]$.

2. Sample $x_{i+1} \sim \pi(x_{i+1} \,|\, y_i)$ by sampling uniformly from the (horizontal) intervals where $\pi(x) > y_i$.

We think of y_i as defining a horizontal 'slice' through the distribution. Slice sampling has multiple advantages over Metropolis-Hastings. The procedure has the opportunity to mix well with respect to sampling from the target distribution, because a horizontal slice may correspond to a large domain that is sampled uniformly, so x_{i+1} can be very far from x_i. In practice, it can be tricky to sample the x_i since doing so in full would require constructing the inverse of $\pi(x)$, but there are various procedures for avoiding this issue while maintaining correctness. Notice also that there is no proposal distribution in slice sampling, which means fewer tuning parameters.

Hybrid Monte Carlo (HMC) introduces an auxiliary 'momentum' variable to embed the action of sampling from the target density $\pi(x)$ within a physical system described by classical mechanics (Duane et al., 1987); it is also called *Hamiltonian Monte Carlo* (Neal, 2010). First, think of $(x, -\pi(x))$ as defining an 'upside down' surface where the original modes of $\pi(x)$ are 'valleys' and low-probability regions are 'uphill.' Now consider a frictionless puck with mass m moving around this surface – its dynamics will be described by its position and its momentum. HMC generates a proposal for a Metropolis algorithm by giving the puck a kick in some direction with some velocity, both random. The puck's trajectory is simulated for some fixed amount of time τ by integrating the system's equations of motion; the final position at time τ is the proposal. This can generate faraway but useful proposals because the puck tends to go downhill toward high-probability regions; it glides over flat equiprobable regions and loses momentum by moving uphill toward low-probability regions.

2.5.2 Ensemble methods

Ensemble (or *population*) methods run multiple chains and accelerate mixing by sharing information between the chains. Examples include affine-invariant ensemble sampling (Goodman and Weare, 2010) and generalized elliptical slice sampling (Nishihara et al., 2014). Below, we focus on a class of ensemble approaches known as *annealing methods.*

Recall that the MH algorithm has trouble sampling from multimodal distributions. Informally, 'flatter' distributions are easier to sample from compared to 'peaky' distributions, especially multimodal ones. Now consider the probabilistic interpretation of a physical multistate system at temperature $\tau > 0$: for a state $x \in \mathcal{X}$, its probability $p(x)$ is proportional to the exponential of the negative of its energy $E(x)$ divided by the temperature, *i.e.*,

$$p(x) \propto \exp(-E(x)/\tau). \tag{2.6}$$

For a given system defined by states and their energies, raising the temperature has the effect of flattening the distribution over those states, while maintaining important features. Annealing methods leverage this intuition to sample more efficiently from difficult targets.

As an example of a popular annealing method, we describe *parallel tempering* (Iba, 2001).[2] Let $\pi(x)$ be the target density over a state space $\mathcal{X}$. The idea is to construct a single Markov chain on the product space $\mathcal{X}^K$ corresponding to an *ensemble* of K Metropolis-Hastings simulations of the system specified by $\pi(x)$ and Eq. 2.6 or its continuous analog, each at a different temperature. Simulations at higher temperatures explore the space more quickly than those at lower temperatures, and they can share information through interactions. One of the K simulations is constructed to marginally have as its stationary distribution the target $\pi(x)$. Explicitly, we can define the system via an energy function of the form $\mathcal{E}(x) = -\log(\pi(x))$.

[2]Following Murray (2007), we cite a review that chronicles the history of parallel tempering.

Now we specify K distributions:

$$\pi_k(x) \propto \exp(-\mathcal{E}(x)c_k), \quad k = 1, \dots K,$$

where c_k can be interpreted as an inverse temperature. Notice that $c_k = 1$ yields $\pi_k(x)$ equal to the target $\pi(x)$, and $c_k = 0$ results in a constant. Thus to obtain K copies of the system, with one equal to the target and the rest at higher temperatures, we can choose the c_k so that $c_1 = 1 > c_2 > c_3 > \cdots > c_K \geq 0$. In each iteration of the algorithm, the K simulations are advanced according to a MH acceptance rule, but they are also allowed to interact, *e.g.*, a pair of simulations may exchange states. Thus, the slower mixing chain indexed by $k = 1$ may jump to states explored by faster mixing chains at higher temperatures. Parallel tempering is popular because its implementation is a straightforward modification to the MH algorithm.

There are several additional classes of annealing methods and other ensemble methods; an excellent review can be found in the PhD thesis by Murray (2007).

2.5.3 Non-reversible methods

The methods described above are representative of the rich menagerie of MCMC algorithms developed using reversible Markov chains where the probability that the chain is in state x and transitions to state x' is equal to the probability that it is in state x' and transitions to x. This condition of detailed balance is straightforward to check, which helps explain the invention of many reversible MCMC methods. Recall that the goal of these methods is to discourage the diffusive behavior of simple Metropolis-Hastings. Intuitively, diffusion is not an efficient mechanism for mixing, say, a cake batter – one uses a spoon or electric mixer to induce a flow that is not equilibrated by a flow in the opposite direction.

Such non-reversibility that discourages 'backtracking' has been difficult to study; a handful of articles describe methods limited to discrete state spaces. These include the theoretical and numerical analysis by Diaconis et al. (2000) of a simple non-reversible chain. The authors start with a reversible unbiased random walk on a one-dimensional finite lattice and then make two copies of the state space, one ‘upstairs’ for transitions to the ‘right’ and one ‘downstairs’ for transitions to the ‘left’, plus transitions between the two levels. This non-reversible chain converges more quickly according to two different distance metrics. Geyer and Mira (2000) reanalyze the same system, this time with respect to asymptotic variance, and find that the most efficient version of the non-reversible chain sweeps through the states in a deterministic way. In a related fashion, Neal (2004) constructs non-reversible chains from reversible chains and demonstrates that their asymptotic variance is no worse than the original reversible chains. Other non-reversible schemes are inspired by non-diffusive physical systems, such as a method for inserting ‘vortices’ by Sun et al. (2010).

2.6 Parallel MCMC

The most obvious way to parallelize MCMC is to run independent simulations in parallel and aggregate their samples. However, this embarrassingly parallel approach does not help to reduce the mixing time, which can be prohibitively long and would be replicated across the parallel instances.

In MCMC, the computational cost is most often determined by the expense of evaluating the target density relative to the mixing time. For example in Metropolis–Hastings, this cost is incurred when the target is evaluated to determine the acceptance ratio of a proposed move. We focus on the increasingly common case where the target is expensive and the dominant computational cost. This evaluation can sometimes be parallelized directly, *e.g.*, when the target function is a product of many individually expensive terms. This sometimes arises in Bayesian inference if the target can be easily decomposed into one likelihood term for each data item. Scalability (*i.e.*, practically achievable speedup) in this setting is limited by the communication and computational costs associated with aggregating the partial evaluations. In general, the target function cannot be parallelized; we divide methods that accelerate MCMC via other sources of parallelism into two classes: parallel ensemble sampling and prefetching.

2.6.1 Parallel ensemble samplers

The ensemble methods discussed earlier run multiple chains that can be simulated in parallel, where any information sharing between chains requires communication. Examples include parallel tempering, described in Section 2.5.2, the emcee implementation (Foreman-Mackey et al., 2012) of affine-invariant ensemble sampling (Goodman and Weare, 2010) and a parallel implementation of generalized elliptical slice sampling (Nishihara et al., 2014).

2.6.2 Prefetching

The second class of parallel MCMC algorithms uses parallelism through speculative execution to accelerate individual chains. This idea is called *prefetching* in some of the literature and appears to have received only limited attention. To the best of our knowledge, prefetching has only been studied in the context of the MH algorithm where, at each iteration, a single new proposal is drawn from a proposal distribution and stochastically accepted or rejected. As shown in Algorithm 2, the body of a MH implementation is a loop containing a single conditional statement and two associated branches. We can thus view the possible execution paths as a binary tree, illustrated in Figure 2.1. The vanilla version of prefetching speculatively evaluates all paths in this binary tree (Brockwell, 2006). The correct path will be exactly one of these, so with J cores, this approach achieves a speedup of $\log_2 J$ with respect to single core execution, ignoring communication and bookkeeping overheads.

Naïve prefetching can be improved by observing that the two branches are not taken with equal probability. On average, the reject branch tends to be more probable; the classic result for the optimal MH acceptance rate is 0.234 (Roberts et al., 1997), so most prefetching scheduling policies have been built around the expectation of rejection. Let $\alpha \leq 0.5$ be the expected acceptance rate. Byrd et al. (2008) introduced *speculative moves*, a procedure that

$$\begin{tikzcd}
& & & & & & & x^t \arrow[dllll] \arrow[drrrr] \\
& & & x_{\mathtt{0}}^{t+1} \arrow[dll] \arrow[drr] & & & & & & & & x_{\mathtt{1}}^{t+1} \arrow[dll] \arrow[drr] \\
& x_{\mathtt{00}}^{t+2} \arrow[dl] \arrow[dr] & & & & x_{\mathtt{01}}^{t+2} \arrow[dl] \arrow[dr] & & & & x_{\mathtt{10}}^{t+2} \arrow[dl] \arrow[dr] & & & & x_{\mathtt{11}}^{t+2} \arrow[dl] \arrow[dr] \\
x_{\mathtt{000}}^{t+3} & & x_{\mathtt{001}}^{t+3} & & x_{\mathtt{010}}^{t+3} & & x_{\mathtt{011}}^{t+3} & & x_{\mathtt{100}}^{t+3} & & x_{\mathtt{101}}^{t+3} & & x_{\mathtt{110}}^{t+3} & & x_{\mathtt{111}}^{t+3}
\end{tikzcd}$$

Figure 2.1: Metropolis–Hastings conceptualized as a binary tree. Nodes at depth d correspond to iteration d, where the root is at depth 0, and branching to the right/left indicates that the proposal is accepted/rejected. Each subscript is a sequence, of length d, of `0`'s and `1`'s, corresponding to the history of rejected and accepted proposals with respect to the root.

speculatively evaluates only along the 'reject' branch of the binary tree; in Figure (2.1), this corresponds to the left-most branch. In each round of their algorithm, only the first k out of $J - 1$ extra cores perform useful work, where k is the number of rejected proposals before the first accepted proposal, relative to the root of the tree. The expected speedup is then:

$$1 + \mathrm{E}(k) < 1 + \sum_{k=0}^{\infty} k(1-\alpha)^k \alpha < 1 + \frac{1-\alpha}{\alpha} = \frac{1}{\alpha}\,.$$

The first term on the left is due to the core at the root of the tree, which always performs useful computation in prefetching schemes. For an acceptance rate of $\alpha = 0.23$, this scheme yields a maximum expected speedup of about 4.3, reaching about 4 with 16 cores, and thus is more limited than the naïve prefetching policy since it essentially cannot take advantage of additional cores. Byrd et al. (2010) later considered the special case where the evaluation of the target occurs on two timescales, slow and fast. This method, called *speculative chains*, modifies speculative moves so that when the target evaluation is slow, available cores are used to speculatively evaluate the subsequent chain, assuming the slow step accepts.

Further extensions to the naïve prefetching scheme allocate cores according to the optimal 'tree shape' with respect to various assumptions about the probability of rejecting a proposal, *i.e.*, by greedily allocating cores to nodes that maximize the depth of speculative computation expected to be correct (Strid, 2010). Next, we summarize Strid's schemes and reference related ideas. *Static prefetching* assumes a fixed acceptance rate; versions of this were proposed earlier in the context of simulated annealing (Witte et al., 1991). *Dynamic prefetching* estimates the acceptance probabilities, *e.g.*, at each level of the tree by drawing empirical MH samples (100,000 in the evaluation), or at each branch in the tree by computing $\min(\beta, \hat{r})$ where β is a constant ($\beta = 1$ in the evaluation) and $\hat{r}$ is an estimate of the MH acceptance ratio based on a fast approximation to the target function. Alternatively, Strid proposes using the approximate target function to identify the single most likely path on which to perform speculative computation. Strid also combines prefetching with other sources of parallelism to obtain a multiplicative effect. To the best of our knowledge, these prefetching methods have been evaluated on up to 64 cores, although usually many fewer.

In the next chapter, we propose *predictive prefetching*, a new scheme that, like Strid's dynamic prefetching, uses an approximation to the target function to predict what computations to prefetch. There are several fundamental differences between our work and Strid's. Most critically, we model the error of the target density approximation, and thus the uncertainty of whether a proposal will be accepted. In addition, we identify a broad class of MCMC algorithms that could benefit from prefetching, not just Metropolis–Hastings, and we show how prefetching can exploit a series of approximations, not just a single one.

2.7 Approximations and large-scale Bayesian inference

Real-world problems are rarely amenable to exact inference, so they require approximate inference in the form of Monte Carlo estimates or *variational* approximations. Unfortunately, approximate Bayesian inference can be challenging when modeling large data sets, as the target posterior density may become expensive to evaluate. This challenge has motivated new methods for inferential computation that can take advantage of approximations to the target density, most often by examining only a subset of the data, or by exploiting closed form approximations such as Taylor series (Christen and Fox, 2005), or by fitting linear or Gaussian process regressions (Conrad et al., 2014).

In Bayesian inference, the target density involves a likelihood, which often decomposes into a product of many factors corresponding to data items, *e.g.*,

$$\pi(\theta \,|\, \mathbf{x}) = \pi_0(\theta)\pi(\mathbf{x} \,|\, \theta) = \pi_0(\theta) \prod_{n=1}^{N} \pi(x_n \,|\, \theta). \tag{2.7}$$

Below, we survey MCMC sampling schemes that exploit this factorization property, motivated by large-scale Bayesian inference with large datasets.

2.7.1 Embarrassingly parallel, approximate MCMC

Several authors have suggested partitioning a large dataset into multiple shards and running MCMC inference on each partition separately across parallel cores. Each of J shards $\{\mathbf{x}^{(j)}\}_{j=1}^{J}$ defines what is sometimes called a *subposterior*:

$$\pi^{(j)}(\theta \,|\, \mathbf{x}^{(j)}) = \pi_0(\theta)^{1/J} \prod_{x \in \mathbf{x}^{(j)}} \pi(x \,|\, \theta), \quad j = 1, \dots, J.$$

The contribution from the original prior is down-weighted so that the original posterior is equal to the product of the J subposteriors, *i.e.*, $\pi(\theta \,|\, \mathbf{x}) = \prod_{j=1}^{J} \pi^{(j)}(\theta \,|\, \mathbf{x}(j))$. However, it is not clear how to combine the samples from the J subposteriors in a coherent fashion to estimate functions of the desired full posterior. Below, we describe three recent efforts.

Neiswanger et al. (2014) explore three potential solutions, ranging from parametric to non-parametric and semi-parametric models. For example, their parametric model invokes the Bayesian central limit theorem; they argue that since a posterior looks like a Gaussian in the limit of many data items, they fit each subposterior with a Gaussian, and then approximate the full posterior as a product of these approximate subposteriors.

Scott et al. (2013) propose *consensus Monte Carlo*, which combines the subposteriors through a weighted average. For Gaussian models, the optimal weight of the jth subposterior is $W_j = \Sigma_j^{-1}$, the inverse of the covariance matrix Σ_j of the subposterior. Assuming a Gaussian model, the authors Monte Carlo estimate Σ_j using the empirical sample variance from the corresponding subposterior.

Finally, Wang and Dunson (2013) propose a *Weierstrass sampler* for parallel MCMC on independent data partitions; these authors provide analytic bounds on the approximation error of their sampler, which appears to be more robust than those described above.

2.7.2 MCMC with mini-batches

Other methods for accelerating MCMC sampling in the case of large-scale Bayesian inference are inspired by stochastic gradient descent. Traditional gradient descent performs optimization by iteratively computing and following a local gradient that depends on a sum of terms corresponding to data items (Dennis and Schnabel, 1983). Stochastic gradient descent is remarkably simple and effective: at each iteration, it uses an approximate gradient based on only a random subset of data, called a *mini-batch*, or even just a single datum (Murata, 1998). Stochastic variational inference techniques adapt these ideas to variational inference (Hoffman et al., 2013), a class of Bayesian procedures that can be efficient but are only approximate in the sense of lacking MCMC's feature of asymptotic correctness.

With MCMC, the idea is to evaluate an approximate posterior whose likelihood term is a noisy estimate based on sampling only one or a few data items. Recent approaches have implemented efficient transition operators that lead to approximate stationary distributions (Welling and Teh, 2011; Ahn et al., 2012; Korattikara et al., 2014; Bardenet et al., 2014; Doucet et al., 2014). Other recent work uses a lower bound on the local likelihood factor to simulate from the exact posterior distribution while evaluating only a subset of the data at each iteration (Maclaurin and Adams, 2014).

In the rest of this thesis, we focus on accelerating MCMC by combining parallelism with approximations to the transition operator through prefetching ideas. Notably, we arrive at a method in which the stationary distribution is *exactly* the target posterior.

Chapter 3

Predictive prefetching with transition operator approximation

We attack the general problem of accelerating MCMC algorithms by using speculative execution to parallelize them. In the previous chapter, our survey of MCMC methods included this approach, sometimes called *prefetching*. An effective prefetching implementation must overcome several challenges, such as correctness. For example, for the results of prefetching to exactly equal those of a serial execution, care is required in the treatment of pseudo-randomness (*i.e.*, each node's source of randomness must produce the same results as it would in a serial execution); slapdash treatment risks introducing biases. But the key challenge for prefetching is performance. A naïve scheduling scheme always requires $\approx 2^J$ parallel cores to achieve a speedup of J. As we saw, this speedup can be improved by leveraging information about the average proposal acceptance rate (Strid, 2010). In particular, if most proposals are rejected, a prefetching implementation can improve its speedup by prefetching more heavily along the reject path. Although in practice the optimal acceptance rate is less than 0.5 (Roberts et al., 1997), extremely small acceptance rates, which lead to good speedup, are accompanied by less effective mixing. If the acceptance rate is set to something like 0.234, speedup is still at most logarithmic.

In this chapter, we propose *predictive prefetching*, a new scheduling approach that uses local information to improve speedup relative to other prefetching schemes. First, we provide a general mathematical framework that allows us to identify a broad class of MCMC algorithms that can benefit from prefetching. Second, we carefully reason about Metropolis–Hastings in a way that maps naturally to prefetching schemes. Next, we describe our predictive prefetching scheme, where we adaptively adjust speculation based not only on the local average proposal acceptance rate – which changes as evaluation progresses – but also on the actual random deviate used at each state. In particular, we describe how we make use of any available fast approximations to the transition operator. Though these approximations are not required, when they are available or learnable, we leverage them to make better scheduling decisions. For the special case of large-scale Bayesian inference, we develop a series of increasingly expensive but more accurate approximations. These decisions are further improved by modeling the error of these approximations, and thus the uncertainty of the scheduling decisions. Performance depends critically on how we model the approximations, and a key insight is in our error model for this setting; much smaller error, and therefore more precise predictions, are obtained by modeling the error of the *difference* between two proposal evaluations, rather than evaluating the errors of the proposals separately. Finally, we provide a theoretical analysis of speedup due to predictive prefetching as a function of predictor accuracy and the number parallel cores. In the next chapter, we describe the details of our system design and implementation, and in the following chapter, we present our actual empirical results.

3.1 Mathematical framework

Consider a transition operator $T(x \to x')$ which has π as its stationary distribution on state space $\mathcal{X}$. Simulation of such an operator typically proceeds using an 'external' source of pseudo-random numbers that can, without loss of generality, be assumed to be drawn uniformly on the unit hypercube, denoted as $\mathcal{U}$. The transition operator is then a deterministic function from the product space of $\mathcal{U}$ and $\mathcal{X}$ back to $\mathcal{X}$, *i.e.*, $T : \mathcal{X} \times \mathcal{U} \rightarrow \mathcal{X}$. Most practical transition operators – Metropolis–Hastings, slice sampling, *etc.* – are actually compositions of two such functions, however. The first function produces a countable set of candidate points in $\mathcal{X}$, here denoted $Q : \mathcal{X} \times \mathcal{U}_Q \rightarrow \mathcal{P}(\mathcal{X})$, where $\mathcal{P}(\mathcal{X})$ is the power set of $\mathcal{X}$. The second function $R : \mathcal{P}(\mathcal{X}) \times \mathcal{U}_R \rightarrow \mathcal{X}$ then chooses one of the candidates for the next state in the Markov chain. Here we have used $\mathcal{U}_Q$ and $\mathcal{U}_R$ to indicate the disjoint subspaces of $\mathcal{U}$ relevant to each part of the operator. In this setup, the basic Metropolis–Hastings algorithm uses $Q(\cdot)$ to produce a tuple of the current point and a proposed point, while multiple-try MH (Liu et al., 2000) and delayed-rejection MH (Tierney and Mira, 1999; Green and Mira, 2001), each create a larger candidate set that includes the current point. In the exponential-shrinkage variant of slice sampling (Neal, 2003), the function $Q(\cdot)$ produces an infinite sequence of candidates that converges to, but does not include, the current point.

This setup is a somewhat more elaborate treatment than usual, but this is intended to serve two purposes: 1) make it clear that there is a separation between generating a set of possible candidates via $Q(\cdot)$ and selecting among them with $R(\cdot)$, and 2) highlight that both of these functions are deterministic functions, given the pseudo-random variates. Others have observed this latter point and used it to construct alternative approaches to MCMC (Propp and Wilson, 1996; Neal, 2012).

We separately consider $Q(\cdot)$ and $R(\cdot)$, because it is generally the case that $Q(\cdot)$ is inexpensive to evaluate and does not require computation of the target density $\pi(x)$, while $R(\cdot)$ must compare the target density at the candidate locations and so represents the bulk of the computational burden. Prefetching MCMC observes that, since $Q(\cdot)$ is cheap and the pseudo-random variates can be produced in any order, the tree of possible future states of the Markov chain can be constructed before any of the $R(\cdot)$ functions are evaluated, as in Figure 2.1 and reproduced here for convenience in Figure 3.1. The sequence of $R(\cdot)$ evaluations simply chooses a path down this tree. We parallelize execution by speculatively choosing to

$$\begin{tikzcd}
 & & & & & & & x^t \arrow[dllll] \arrow[drrrr] & & & & & & & \\
 & & & x_{\mathtt{0}}^{t+1} \arrow[dll] \arrow[drr] & & & & & & & & x_{\mathtt{1}}^{t+1} \arrow[dll] \arrow[drr] & & & \\
 & x_{\mathtt{00}}^{t+2} \arrow[dl] \arrow[dr] & & & & x_{\mathtt{01}}^{t+2} \arrow[dl] \arrow[dr] & & & & x_{\mathtt{10}}^{t+2} \arrow[dl] \arrow[dr] & & & & x_{\mathtt{11}}^{t+2} \arrow[dl] \arrow[dr] & \\
x_{\mathtt{000}}^{t+3} & & x_{\mathtt{001}}^{t+3} & & x_{\mathtt{010}}^{t+3} & & x_{\mathtt{011}}^{t+3} & & x_{\mathtt{100}}^{t+3} & & x_{\mathtt{101}}^{t+3} & & x_{\mathtt{110}}^{t+3} & & x_{\mathtt{111}}^{t+3}
\end{tikzcd}$$

Figure 3.1: Metropolis–Hastings conceptualized as a binary tree. Each level of the tree represents an iteration, where branching to the right/left indicates that the proposal is accepted/rejected. Each subscript is a sequence of `0`'s and `1`s corresponding to the history of rejected and accepted proposals with respect to the root. (Reproduced from Figure 2.1.)

evaluate $R(\{x_i\}, u)$ for some parts of the tree that have not yet been reached. If one or more nodes in this subtree are eventually reached, then we achieve a speedup.

For clarity, in the remainder of this thesis we focus on the straightforward random-walk Metropolis–Hastings operator; we depict our view of its simulation in Algorithm 3. In this special case, $Q(\cdot)$ produces a tuple of the current point and a proposal. The function $R : \mathcal{X} \times \mathcal{X} \times (0, 1) \to \mathcal{X}$ takes these two points, along with a uniform random variate in $(0, 1)$, and selects one of the two inputs via:

$$R(x, x', u) = \begin{cases} x' & \text{if } u\dfrac{q(x' \,|\, x)}{q(x \,|\, x')} < \dfrac{\pi(x')}{\pi(x)} \\ x & \text{otherwise} \end{cases}, \tag{3.1}$$

where $q(\cdot \,|\, \cdot)$ is the proposal density corresponding to $Q(\cdot)$. We write the acceptance ratio in this somewhat unusual fashion to highlight the fact that the left-hand side of the inequality does not require evaluation of the target density and is easy to precompute.

Algorithm 3 Our view of Metropolis–Hastings.

```
Input: Initial state $x^0$, number of iterations $T$, target $\pi(x)$, proposal $q(x' \mid x)$
Output: Samples $x^1, \dots, x^T$
for $t$ in $0, \dots, T-1$ do
    $\mathbf{u}_Q^t = \{u_{Q,i}^t \sim \mathrm{Unif}(0,1)\}$    ▷ Pseudo-random numbers consumed by $Q(\cdot)$
    $u_R^t \sim \mathrm{Unif}(0,1)$    ▷ Pseudo-random number for $R(\cdot)$
    $(x^t, x') \leftarrow Q(x^t, \mathbf{u}_Q^t) = (x^t, x' \sim q(x' \mid x))$    ▷ Produce two candidates
    $x^{t+1} \leftarrow R(x^t, x', u_R^t) = \begin{cases} x' & \text{if } u_R^t \frac{q(x' \mid x^t)}{q(x^t \mid x')} < \frac{\pi(x')}{\pi(x^t)} \\ x^t & \text{otherwise} \end{cases}$    ▷ Select next state
end for
```

3.2 Metropolis–Hastings simulation

In this section, we reason about Metropolis–Hastings simulation through the lens of the binary state tree. This enables us to coherently reason about prefetching schemes as well as motivate and describe our approach in the next section. First, we develop some notation that gives us a language for talking about the MH tree. This notation will also map to the data structures and routines used in our system design, described in the next chapter. Now, recall that prefetching schemes use parallel cores to precompute the target density at states that might be considered during simulation. Thus, we next use the tree to identify where computation occurs with respect to a particular simulation and then discuss the use of pseudo-randomness with respect to the tree. Finally, we introduce a new binary tree, the *jobtree*, that simplifies how to reason about computation during MH simulation; this will help us cleanly describe our approach in the next section and will form the central data structure of our system in the next chapter.

3.2.1 Bit string notation

We use small Greek letters (α, β) for elements of $\{0,1\}^*$. Let ϵ be the empty string. Given a bit string α, let $\lfloor \alpha \rfloor$ equal α with all trailing `0` bits removed. Define flip(α) as α with the

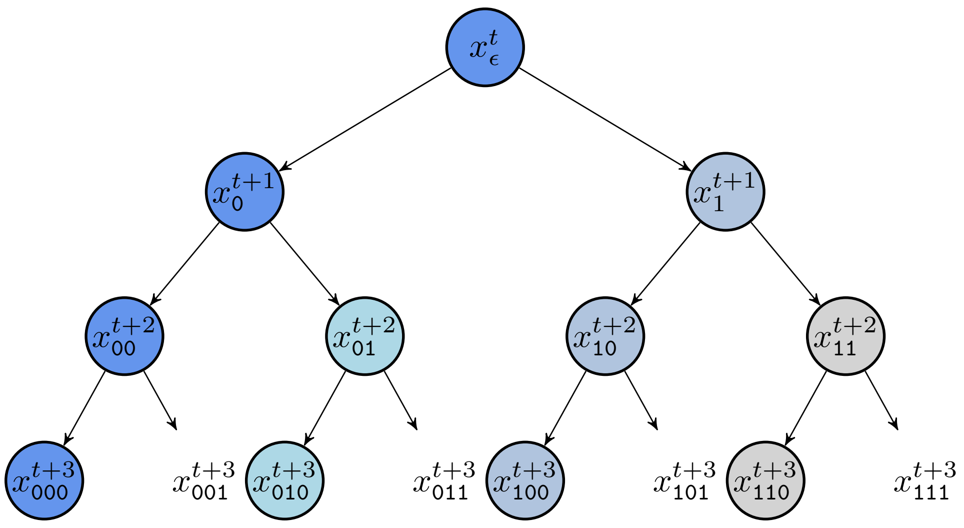


Figure 3.2: Metropolis–Hastings conceptualized as a binary tree. Each level of the tree represents an iteration, where branching to the right/left indicates that the proposal is accepted/rejected. Each subscript is a sequence of $\texttt{0}$'s and $\texttt{1}$'s corresponding to the history of rejected and accepted proposals with respect to the current state x^t at the root. Nodes of the same color correspond to sequences of states that are equal, which happens when proposals are rejected. For example, the nodes along the left-most branch are all equal to the root and correspond to a sequence of three rejected proposals. The four uncolored nodes at the bottom of the tree represent the possible proposals at iteration $t+3$ and are distinct.

last bit flipped, as follows:

$$\text{flip}(\alpha) = \begin{cases} \texttt{1} & \text{if } \alpha = \epsilon, \\ \beta\texttt{1} & \text{if } \alpha = \beta\texttt{0}, \\ \beta\texttt{0} & \text{if } \alpha = \beta\texttt{1}. \end{cases}$$

3.2.2 Mapping states to bit strings

Recall from Figure 3.1 that we can conceptualize Metropolis–Hastings as a binary tree: given the current state, the possible sequences of future states result from accepting or rejecting a proposal at each iteration. We reproduce this tree in Figure 3.2, this time with different colors to highlight sequences of possible Markov chain states that are identical, due to sequences of rejected proposals. Each node is labeled with a distinct subscript, mapping each possible state to a bit string that records the history of the chain, as we describe below.

Without loss of generality, call the current state x^0. Let iteration t simulate the transition from a state x^t to the next state x^{t+1}, as in our descriptions of Metropolis–Hastings in Algorithms 2 and 3. For all $t \geq 0$, define a mapping $\rho(x^t) \equiv \rho^t$ that identifies with each possible state x^t a bit string ρ^t, relative to the current state x^0, as follows:

$$\rho(x^t) = \begin{cases} \epsilon & \text{if } t = 0, \\ \rho^{t-1}\mathtt{1} & \text{if } t > 0 \text{ and proposal at iteration } t \text{ is accepted,} \\ \rho^{t-1}\mathtt{0} & \text{otherwise, in which case the proposal is rejected and } x^t = x^{t-1}. \end{cases}$$

In other words, the current state x^0 is mapped to ϵ and otherwise, x^t is mapped to a sequence of `0`'s and `1`'s corresponding to its history of rejected and accepted proposals, respectively. The length of ρ^t is t and ρ^t is a prefix of ρ^T for all $T \geq t$. Note that an inverse mapping from bit strings to states, *i.e.*, $\rho^{-1}(\rho^t) = x^t$, must satisfy $\rho^{-1}(\alpha) = \rho^{-1}(\lfloor\alpha\rfloor)$. This corresponds to the fact that sequences of rejected proposals yield sequences of Markov chain states equal to either the last accepted proposal or the initial state, if no such proposal exists.

3.2.3 Computation with respect to a simulation path

Figure 3.3 depicts one instance of a Metropolis–Hastings simulation superimposed on the binary tree of all possible states. As before, left and right children correspond to the state after a proposal has been rejected or accepted, respectively. Given the current state at the root, the states of the simulated Markov chain correspond to a single connected path through the tree. We call this the *simulation path.*

Each iteration simulates one MH transition and involves evaluating the target density at a new proposal. A node corresponding to a rejected proposal is *not* directly on the simulation path, but its left sibling and parent as well as other ancestors are all on the simulation path. The state at the left sibling is equal to the state at the parent. Thus, the MH algorithm involves computations at and only at three kinds of nodes: the root, nodes on the simulation path that are right children (accepted proposals) and the right siblings of nodes on the

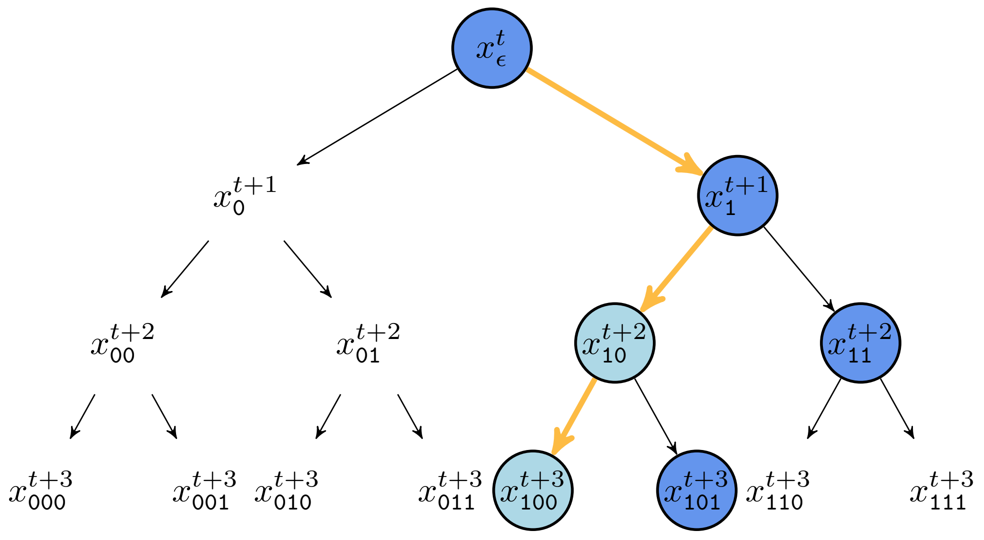


Figure 3.3: Schematic of a Metropolis–Hastings simulation superimposed on the binary tree of all possible states. As in Figure 3.1, each level of the tree represents an iteration, where branching to the right/left indicates that the proposal is accepted/rejected. The simulation path (thick arrows) starts at the root x^t and connects the output states: $x_1^{t+1}, x_{10}^{t+2}, x_{100}^{t+3}$. In this example, the first proposal is accepted and the next two proposals are rejected. The dark filled circles indicate states where the target density is evaluated during simulation. Those that are not on the simulation path correspond to rejected proposals. Their siblings are pale filled circles on the simulation path; since each of the corresponding states is a copy of its parent, its target density does not need to be reevaluated during the subsequent comparison to the next proposal.

simulation path that are left children (rejected proposals).

3.2.4 Using pseudo-randomness

For a particular transition operator $T(x \to x')$ as described in Section 3.1, the number of random variates required to simulate one transition in general depends on the starting state x. For example, in Metropolis–Hastings as described in Algorithm 3, iteration t consumes at least one random variate $\mathbf{u}_Q^t$ to generate the proposal plus exactly one random variate u_R^t to select the next state. The MH proposal step can easily consume a non-constant number of random variates, *e.g.*, if the proposal is generated via rejection sampling, as is common when dealing with truncated distributions.

This subtle point matters when thinking about prefetching schemes as it implies that the consumption of a pseudo-random stream during Markov chain simulation depends on the history of the chain. With respect to the MH tree as illustrated in Figure 2.1, this means that a pseudo-random stream may be consumed at different rates, depending on what simulation path is taken on the tree. From our reading of prior work on prefetching, it is not clear to us whether this issue has been addressed or ignored; *e.g.*, Strid (2010) casually refers to the handling of pseudo-random numbers in prefetching schemes as “an implementation issue.”

In our use of prefetching, given an initial state and an initialized pseudo-random stream, we require the simulated chain to be *equal* to that produced by a serial execution, not merely statistically equivalent. To satisfy this constraint, there are several strategies for managing the pseudo-random stream so that its use with prefetching equals that during serial execution. The first is to synchronize the use of the stream across all possible simulation paths so that the sequence of random variates available at iteration t depend only on t. In the language of the MH tree, the random variates used to simulate the transition from a node at depth t to $t + 1$ are shared across all possible transitions at this layer in the tree. This can be achieved by reseeding a random number generator at the start of each iteration, *e.g.*, using the random variates of a separate pseudo-random stream as the sequence of seeds. Alternatively, if k gives an upper bound on the number of random variates consumed at each iteration, then the stream can be allocated across iterations so that iteration t is constrained to use the k random variates starting at the kt-th location in the stream. A jump-ahead random number generator, could be useful for such a scheme, *e.g.*, the algorithm by Haramoto et al. (2008a,b). The final strategy – which is the one we follow in our implementation, described in the next chapter – is to use the pseudo-random stream exactly as in a ‘normal’ serial execution. This leads to history-dependent consumption of the random variates and requires a small amount of bookkeeping.

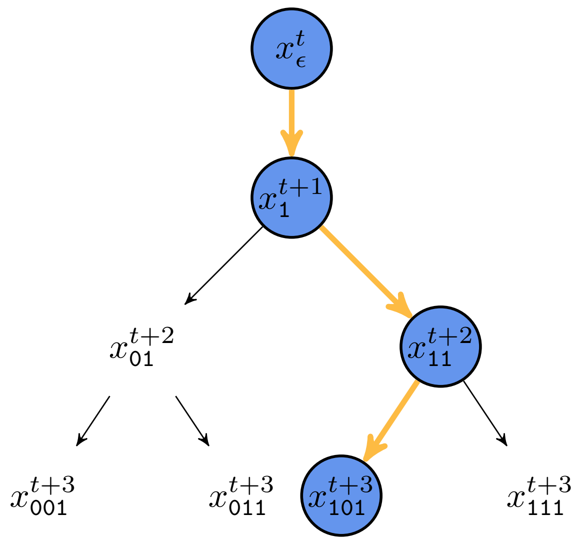


Figure 3.4: Schematic of the same Metropolis–Hastings simulation as in Figure 3.3, this time superimposed on the jobtree. Recall that, given the current state x_ϵ^t, the simulated chain in this example is: $x_1^{t+1}, x_{10}^{t+2}, x_{100}^{t+3}$, corresponding to one accept followed by two rejects. This tree includes only those nodes in the original MH tree where a new state is introduced and thus the target density must be evaluated when comparing such a state to another. States where the the target density is evaluated in a serial MH execution (filled circles) are now connected by a single path (thick arrows) that we call a computation path.

3.2.5 Representing computation with the jobtree

Here, we introduce the Metropolis–Hastings *jobtree*, depicted in Figure 3.4. Like the original MH state tree, the jobtree is generally binary, except that the root has only one child. It contains all of the same information as the MH state tree yet is more compact as it represents only those states where new computation occurs, *i.e.*, where the target density must be evaluated in order to compare such a state to another. Specifically, it includes the root node and all right children of the MH state tree, corresponding to the current state and all possible subsequent proposals – together, these specify the possible distinct states and at what iteration each would first appear. Since the jobtree leaves out all left children – which are equal to their parents – it includes about half as many nodes as the MH state tree.

While the nodes in the jobtree are a subset of the nodes in the MH tree, the jobtree itself is not subtree of the MH tree. In the jobtree, the root has one out-edge that represents the *immediate comparison* between the current state and corresponding proposal, which must occur at the current iteration. We refer to the transition from the current state to the next state as the *immediate transition*. The nodes below the root are all proposals and each has two children: the left child corresponds to the next proposal if its parent proposal is rejected, and the right child corresponds to the next proposal if its parent is accepted.

Recall that in the MH tree, simple paths correspond to instantiations of simulated Markov chains but do not capture all nodes where computation occurs. Paths on the MH jobtree represent computation in the sense that they map to sequences of states where the target density is evaluated during serial MH simulation. We refer to any such path as a *computation path*; an example is shown in Figure 3.4.

Recall that the MH transition operator selects between two states; in the MH tree, two such states are represented as siblings. For any pair of sibling nodes in the MH tree, there is an equivalent pair of nodes in the jobtree. We can see this by first considering the MH tree, and recalling that the state at any left child is equal to its parent and more generally to all ancestors that are also left children. Consider a proposal, *i.e.*, some right child in the MH tree, encoded by a bit string ρ. Its left sibling is encoded as $\text{flip}(\rho)$ and the oldest ancestor whose state is equal to the left sibling as $\lfloor \text{flip}(\rho) \rfloor$. In deciding whether to accept a proposal, the MH transition operator compares the proposal x_ρ to a state equal to $x_{\lfloor \text{flip}(\rho) \rfloor}$, which we call its *comparison parent*. In Figure 3.5, we draw back-edges from each proposal to its comparison parent. A comparison parent is always either the root node or a right child corresponding to a proposal, and thus is also in the jobtree. We illustrate this in Figure 3.6 by adding these back-edges from each proposal in the jobtree to its comparison parent.

3.2.6 Metropolis–Hastings with prefetching

A prefetching framework schedules cores to simulate the immediate transition and prefetch possible future transitions. This scheduling could be performed at many levels of granularity; for concreteness and simplicity, let us map cores to transitions. Then, a prefetching framework with J cores uses one core to simulate the immediate transition and the others to precompute

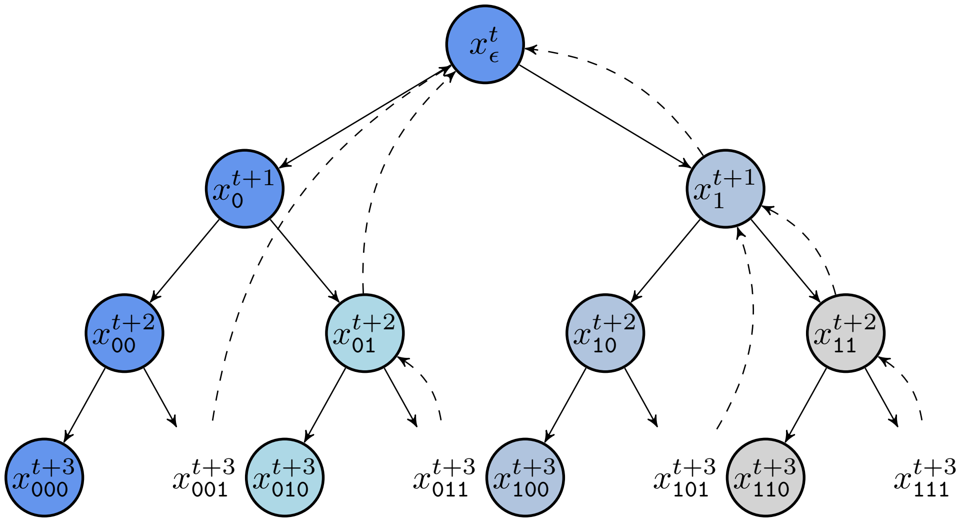


Figure 3.5: The Metropolis–Hastings binary state tree. As in Figure 3.2, nodes of the same color correspond to sequences of states that are equal, which happens when proposals are rejected. Here, we add dashed back-edges from a proposal node ρ to its comparison parent $\lfloor \text{flip}(\rho) \rfloor$, the oldest ancestor equal to its left sibling. Each comparison parent is either the root node or a right child, and so is also a node in the jobtree, depicted in Figure 3.6.

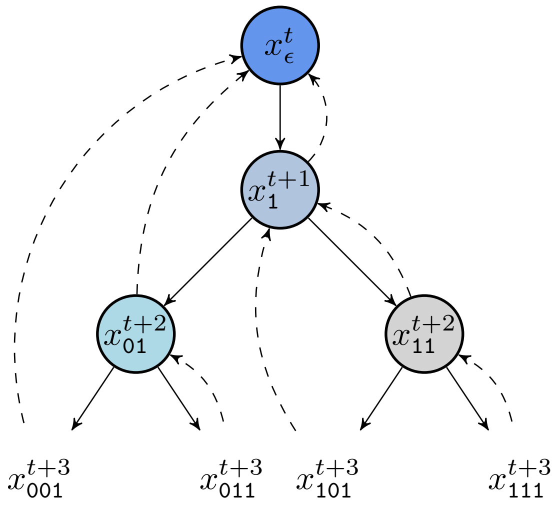


Figure 3.6: The Metropolis–Hastings jobtree, here with back-edges that connect each proposal node ρ to its comparison parent $\lfloor \text{flip}(\rho) \rfloor$. For any proposal, the state of its comparison parent is equal to that of its left sibling, and so these back-edges represent the comparison that is made in evaluating whether the proposal is accepted. The colored nodes correspond to those as in Figures 3.2 and 3.5 of the MH tree, where each group of nodes with the same color share the same state. In the jobtree, the states are distinct and so there is only one node for each color.

transitions for possible future iterations. The actual set of transitions that must be simulated maps to a single computation path on the jobtree. If each precomputation falls along the actual computation path, the framework will achieve the ideal linear speedup (evaluating T iterations will take time proportional to T/J). If some of them do not fall along the chain, the framework will fail to scale perfectly with the available resources. For instance, a naïve framework that evaluates transitions based on breadth-first search of the prefetching state tree (Figure 2.1) will achieve logarithmic speedup (time proportional to $T/\log_2 J$). Good speedup thus is possible with prefetching, if we can make good predictions about which computation path will be taken on the jobtree. This is in turn determined by the ability to predict whether the MH threshold will be exceeded in Equation 3.1.

3.3 Predictive prefetching: Exploiting predictions

In this section, we propose *predictive prefetching*, a principled scheduling approach that exploits predictions about whether possible precomputations will fall along the true computation path. Let ρ denote a node on the tree, x_ρ indicate the current state at ρ, and x'_ρ indicate the proposal. Note that in the language of the jobtree, $x_\rho = x_{\lfloor \text{flip}(\rho) \rfloor}$ is the comparison parent of x'_ρ. For convenience, let us define

$$\gamma_\rho = u_\rho \frac{q(x'_\rho \,|\, x_\rho)}{q(x_\rho \,|\, x'_\rho)} \tag{3.2}$$

where u_ρ is the MH threshold variate associated with node ρ. The Markov chain's steps are determined by iterations of computing the indicator function

$$\iota_\rho = \mathbb{I}(\gamma_\rho < \pi(x'_\rho)/\pi(x_\rho)), \tag{3.3}$$

where a proposal is accepted iff $\iota_\rho = 1$. The quantities x_ρ, x'_ρ, and γ_ρ can be inexpensively computed at any time from the stream of pseudo-random numbers, without examining the expensive target $\pi(\cdot)$.

The precomputation schedule should maximize expected speedup, which corresponds to the expected number of precomputations along the true computation path in the jobtree. To maximize this quantity, the framework needs to anticipate which branches of the jobtree are likely to be taken. The root node and its only child are always evaluated. We associate with each remaining node ρ in the jobtree a predictor ψ_ρ that models the conditional probability that x_ρ is accepted, given that ρ is on the computation path. If ρ is on the computation path and x_ρ is accepted/rejected, then its right/left child will be the next node on the true computational path. In Figure 3.7, we illustrate the predictor with respect to the jobtree by labeling the edges with *branch probabilities*: the edge from a node ρ to its right child has branch probability equal to the predictor ψ_ρ and the edge to its left child has branch probability $1 - \psi_\rho$. Note that a predictor ψ_ρ may vary over time. When the target functions $\pi(x_\rho)$ and $\pi(x'_\rho)$ are completely evaluated, we require that the predictor ψ_ρ equals the indicator ι_ρ in Equation 3.3. We use the predictors to model the *expected utility* of a node ρ, *i.e.*, the probability that ρ is on the computation path. This is given by the product of the branch probabilities along the path connecting the root to ρ, as we illustrate in Figure 3.8. We take a *greedy* approach, scheduling those nodes with maximum expected utility for precomputation.

A predictor is always available – *e.g.*, one can use the recent acceptance probability; Figures 3.7 and 3.8 take as an example $\psi_\rho = 0.2$. Alternatively, suppose we have access to a fast approximation to the target density $\tilde{\pi}(x)$ and model the error of approximately evaluating $\log(\pi(x'_\rho)/\pi(x_\rho))$ as normally distributed with variance σ^2. Then we can write:

$$\psi_\rho = \Pr\left(\log\gamma_\rho < \log\left(\frac{\pi(x'_\rho)}{\pi(x_\rho)}\right) \;\middle|\; \tilde{\pi}(x), \sigma^2\right) \tag{3.4}$$

$$= \int_{\log\gamma_\rho}^{\infty} \mathcal{N}\left(z \;\middle|\; \log\left(\frac{\tilde{\pi}(x'_\rho)}{\tilde{\pi}(x_\rho)}\right), \sigma^2\right) dz. \tag{3.5}$$

More generally, we can often improve predictions using computation. To model this, we

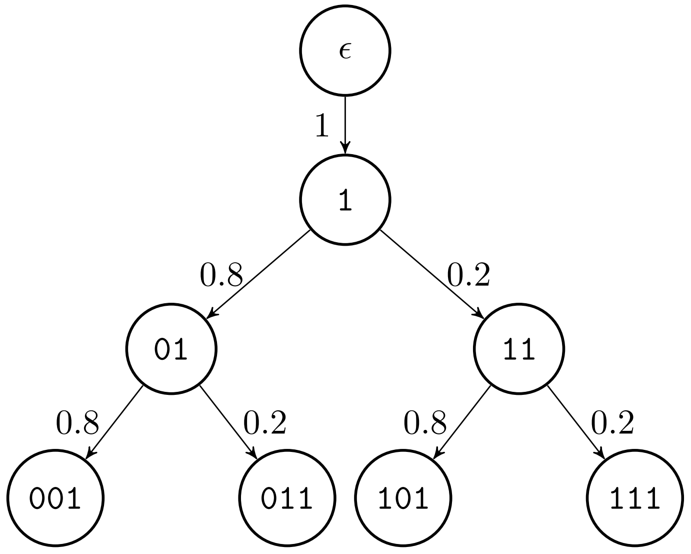


Figure 3.7: Example Metropolis–Hastings jobtree, as in Figure 3.4, here with edges labeled by their branch probabilities and nodes labeled by their bit strings. We associate with each node ρ a predictor ψ_ρ that models the conditional probability that x_ρ is accepted, given that ρ is on the computation path. The edge from a node ρ to its right child has branch probability equal to the predictor ψ_ρ and the edge to its left child has branch probability $1 - \psi_\rho$. This example illustrates the branch probabilities for a predictor ψ_ρ based solely on an average acceptance rate of 0.2.

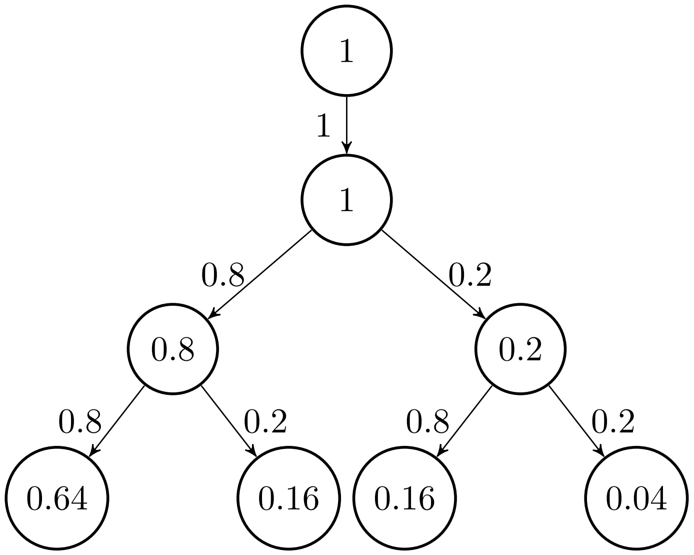


Figure 3.8: Example Metropolis–Hastings jobtree where each node ρ is labeled by the probability that ρ is on the computation path. As in Figure 3.7, an edge pointing to a node ρ is labeled by the branch probability indicating the belief that node ρ will be on the computation path, given that its parent is on the computation path. The probability that a node ρ is on the computation path then equals the product of the branch probabilities along the path from the root to ρ.

define a sequence of estimators

$$\psi_\rho^{(m)} \approx \psi_\rho, \quad m = 0, 1, 2, \dots, N, \tag{3.6}$$

where increasing m implies increasing expected accuracy, and $\psi_\rho^{(N)} = \iota_\rho$. Workers move through this sequence until they perform the exact computation. The predictor sequence affects scheduling decisions: once it becomes sufficiently certain that a worker's branch will not be taken, that worker and other workers computing on its descendent nodes should be reallocated to more promising branches. Ultimately, every step that is actually taken on the Markov chain is computed to completion. The approach simulates from the true stationary distribution, not an approximation thereof. The estimators are used only to make scheduling decisions for prefetching.

There are several schemes for producing this estimator sequence, and predictive prefetching applies to any Markov chain Monte Carlo problem for which approximations are available. We focus on the important case where the target density is a posterior and the likelihood depends on a (possibly large) dataset. Specifically, we obtain a fast approximation to the posterior by estimating the likelihood with only a subset of the data, and improve estimates by including more and more data.

3.4 An estimator for large-scale Bayesian inference

In Bayesian inference with MCMC, the target density is a (possibly unnormalized) posterior distribution. In most modeling problems, the target density can be decomposed into a product of terms. If the data $\mathbf{x} = \{x_n\}_{n=1}^N$ are conditionally independent given the model parameters θ, there is a factor for each of the N data:

$$\pi(\theta \,|\, \mathbf{x}) \propto \pi_0(\theta)\, \pi(\mathbf{x} \,|\, \theta) = \pi_0(\theta) \prod_{n=1}^{N} \pi(x_n \,|\, \theta). \tag{3.7}$$

Here $\pi_0(\theta)$ is a prior distribution and $\pi(x_n \,|\, \theta)$ is the likelihood term associated with the nth datum. The logarithm of the target distribution is a sum of terms,

$$\mathcal{L}(\theta) = \log \pi(\theta \,|\, \mathbf{x}) = \log \pi_0(\theta) + \log \pi(\mathbf{x} \,|\, \theta) + c = \log \pi_0(\theta) + \sum_{n=1}^{N} \log \pi(x_n \,|\, \theta) + c \,, \tag{3.8}$$

where c is an unknown constant that does not depend on θ and can be ignored. Our predictive prefetching algorithm uses this to form predictors ψ_ρ as in Equation 3.5. We can reframe ψ_ρ using log probabilities as

$$\psi_\rho \approx \Pr\left(\log \gamma_\rho < \mathcal{L}(\theta') - \mathcal{L}(\theta)\right), \tag{3.9}$$

where γ_ρ is the precomputed random MH threshold of Equation 3.2. One approach to forming this predictor is to use a normal model for each $\mathcal{L}(\theta)$, as done by Korattikara et al. (2014). However, rather than modeling $\mathcal{L}(\theta)$ and $\mathcal{L}(\theta')$ separately, we can achieve a better estimator with lower variance by considering them together. Expanding each log likelihood gives:

$$\mathcal{L}(\theta') - \mathcal{L}(\theta) = \log \pi_0(\theta') - \log \pi_0(\theta) + \sum_{n=1}^{N} \Delta_n \tag{3.10}$$

$$\Delta_n = \log \pi(x_n \,|\, \theta') - \log \pi(x_n \,|\, \theta) \,. \tag{3.11}$$

In Bayesian posterior sampling, the proposal θ' is usually a perturbation of θ and so we expect $\log \pi(x_n \,|\, \theta')$ to be correlated with $\log \pi(x_n \,|\, \theta)$. In this case, the differences Δ_n occur on a smaller scale than they would otherwise and also have a smaller variance compared to the variance of $\log \pi(x_n \,|\, \theta)$ across data terms.

A concrete sequence of estimators is obtained by subsampling the differences. Let $\{\Delta_n\}_{n=1}^{m}$ be a subsample of size $m < N$, without replacement, from $\{\Delta_n\}_{n=1}^{N}$. This subsample can be used to construct an unbiased estimate of $\mathcal{L}(\theta') - \mathcal{L}(\theta)$. We model the terms of this subsample as i.i.d. from a normal distribution with bounded variance σ^2, leading to:

$$\mathcal{L}(\theta') - \mathcal{L}(\theta) \sim \mathcal{N}(\hat{\mu}_m, \hat{\sigma}_m^2) \,. \tag{3.12}$$

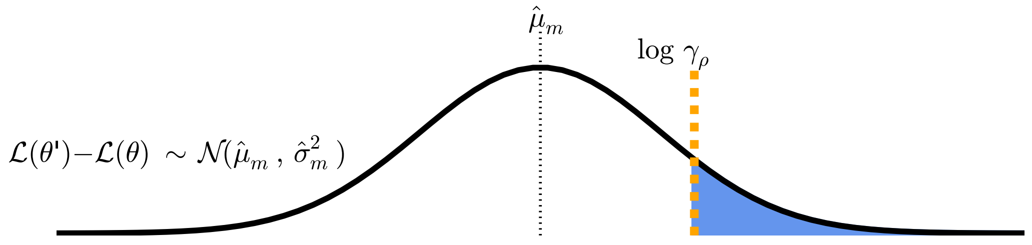

$\hat{\mu}_m$
$\log\gamma_\rho$
$\mathcal{L}(\theta')-\mathcal{L}(\theta) \sim \mathcal{N}(\hat{\mu}_m, \hat{\sigma}_m^2)$

Figure 3.9: We use a normal model for the difference of log posteriors at states θ' and θ as $\mathcal{L}(\theta') - \mathcal{L}(\theta) \sim \mathcal{N}(\hat{\mu}_m, \hat{\sigma}_m^2)$. Thus, the predictor $\psi_\rho^{(m)}$ is equal to the area under $\mathcal{N}(\mu, \sigma^2)$ to the right of $\log\gamma_\rho$. Recall that γ_ρ is the precomputed random MH threshold of Equation 3.2 and depends on a uniform random variate $u \sim \text{Unif}(0,1)$.

The mean estimate $\hat{\mu}_m$ is empirically computable:

$$\hat{\mu}_m = \log\pi_0(\theta') - \log\pi_0(\theta) + \frac{N}{m}\sum_{n=1}^{m}\Delta_n\,. \tag{3.13}$$

The error estimate $\hat{\sigma}_m$ may be derived from $s_m/\sqrt{m}$, where s_m is the empirical standard deviation of the m subsampled Δ_n terms. To obtain a confidence interval for the sum of N terms, we multiply this estimate by N and the finite population correction $\sqrt{(N-m)/N}$, giving:

$$\hat{\sigma}_m = s_m\sqrt{\frac{N(N-m)}{m}}\,. \tag{3.14}$$

As illustrated in Figure 3.9, we can now form the predictor $\psi_\rho^{(m)}$ by considering the tail probability for $\log\gamma_\rho$, where recall γ_ρ is defined in Equation 3.2:

$$\begin{aligned}
\psi_\rho^{(m)} &= \int_{\log\gamma_\rho}^{\infty} \mathcal{N}(z \,|\, \hat{\mu}_m, \hat{\sigma}_m^2)\,\mathrm{d}z & (3.15)\\
&= 1 - \int_{-\infty}^{\log\gamma_\rho} \mathcal{N}(z \,|\, \hat{\mu}_m, \hat{\sigma}_m^2)\,\mathrm{d}z \\
&= \frac{1}{2}\left[1 - \operatorname{erf}\left(\frac{\log\gamma_\rho - \log\hat{\mu}_m}{\sqrt{2}\hat{\sigma}_m}\right)\right] \\
&= \frac{1}{2}\left[1 + \operatorname{erf}\left(\frac{\log\hat{\mu}_m - \log\gamma_\rho}{\sqrt{2}\hat{\sigma}_m}\right)\right]\,. & (3.16)
\end{aligned}$$

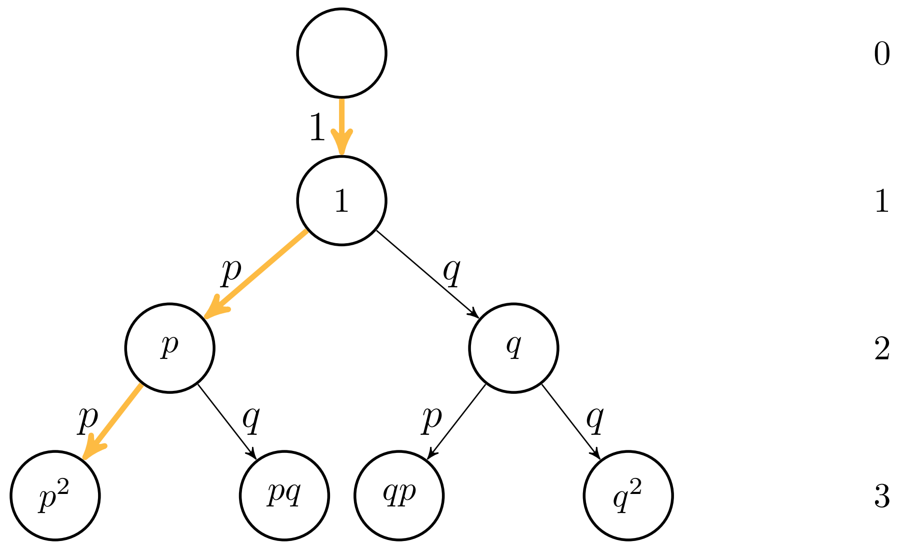


Figure 3.10: Metropolis–Hastings jobtree in our biased coin model. Nodes are arranged so that the left-most branch is the (unknown) true computation path (thick arrows). Furthermore, given a node, the predictor expects the left (right) child to be next on the path with probability p (q), where $p \geq q$ and $p + q = 1$. Each node is labeled by the probability the predictor expects it to be on the true path. The root is complete; workers are scheduled to the remaining nodes greedily. We index depth in the tree from 0 at the root.

3.5 Speedup with instantaneous, imperfect predictions

If we could make instantaneous, perfect predictions, then predictive prefetching would achieve perfect speedup, in an ideal system with zero communication costs or other overheads. In reality, we have access only to imperfect predictions, and we use probabilistic models to characterize our uncertainty about these predictions. In this section, we analyze the expected speedup of predictive prefetching, as a function of predictor accuracy, for infinitely fast predictions in an ideal system.

Consider a specific MH simulation of fixed length. Suppose we have access to a predictor ψ_ρ that models the conditional probability that x_ρ is accepted, given that ρ is on the computation path, as introduced in Section 3.3. We model the predictor's accuracy by a biased coin, depicted in Figure 3.10. Let p be the probability that the expected outcome is the true outcome and let $q = 1 - p$ be the corresponding probability that it is not. We can think of p and q as the accuracy and error, respectively, of the predictor. Let τ_J denote the

MH simulation running time, using predictive prefetching with J workers. Then, $S_J = \tau_1/\tau_J$ is the speedup relative to a single worker. Our objective is to understand S_J as a function of q and J.

In the limit of perfect predictions, $q = 0$ and predictive prefetching obtains perfect, linear speedup, with $S_J = J$. In the limit of uninformative predictions, $q = 1/2$ and predictive prefetching reduces to the naïve scheme, leading to logarithmic speedup. The expected speedup is $\log_2(J+1)$, *e.g.*, $\mathrm{E}[S_1] = 1$, $\mathrm{E}[S_3] = 2$, $\mathrm{E}[S_7] = 3$. Below, we analyze the expected speedup for imperfect predictions, where $0 < q < 1/2$. We do not study the adversarial scenario of malicious predictions, where $q > 1/2$, which happens when we believe our predictors to be informative, but they are actually incorrect on average.

To calculate expected speedup, we need to understand how the greedy scheme, described in Section 3.3, schedules workers on the jobtree. For simplicity, we consider one 'round' of the algorithm, initialized as follows: the jobtree is known to depth J, where the root is considered depth 0, the target has been evaluated at the root and nowhere else; at all other nodes, only the predictor has been evaluated. We break down our analysis into two parts. First, we consider the scheduled workers' depth, which gives us a lower bound on the expected speedup. Then, we give a complete description of the workers' allocation, which allows us to calculate the expected speedup. Finally, we consider a scheme that combines predictive prefetching with parallel computation at each node.

3.5.1 Worker depth and simple bounds on speedup

In the limit of perfect predictions, $q = 0$, and predictive prefetching schedules workers along the true computation path in the jobtree. When $0 < q < 1/2$, our biased coin model assigns the k-th node along the true path a probability of p^k, where the root is indexed by $k = 0$. In Figure 3.10, this corresponds to the left-most branch. As long as $p^k > pq$, a greedy scheduling algorithm places at least k cores along this path before starting to consider alternate paths. Let K denote the maximum value of k before the greedy algorithm starts to allocate cores away from a single path, *i.e.*, K is the largest value of k satisfying $p^k = (1-q)^k \geq q$. Then,

$$K = \left\lfloor \frac{\log q}{\log(1-q)} \right\rfloor. \tag{3.17}$$

For example, when $q = 0.1, 0.01, 0.001$, then $K = 21, 458, 6904$, respectively. Figure 3.11 plots K as a function of q on a log-log scale. With K cores, the expected speedup is the sum of the probabilities of these nodes, giving us the following lower bound:

$$\mathrm{E}[S_K] \geq \sum_{k=1}^{K} p^k = -1 + \sum_{k=0}^{K} p^k = -1 + \frac{1-p^{K+1}}{1-p} = \frac{p - p^{K+1}}{1-p} = \frac{(1-q)-(1-q)^{K+1}}{q}.$$

Since K tells us about the depth of the tree, it also gives us an upper bound on S_J. With $J \leq K$ cores, they are all placed along the left branch, thus $S_J < J$. With $J > K$ cores, for reasonable values of J and q, workers are allocated to other nodes at depths no greater than K, thus $S_J < K$. Figure 3.12 depicts these lower and upper bounds on the expected speedup, as a function of J, for different values of q.

3.5.2 Worker allocation and expected speedup

The greedy algorithm places J cores at the J nodes with the highest probabilities of occurring on the true path. Let us encode a node at depth i by a bit string $\gamma \in \{0,1\}^i$, where the root is at depth 0. In our encoding introduced in Section 3.2.1, `1` (`0`) denotes that, given a state θ, the proposal θ' is accepted (rejected). Here, let `1` (`0`) denote that a prediction based on the expected outcome is correct (incorrect) with probability p ($q = 1-p$).

Let $\Pr(\gamma \,|\, q)$ denote the probability that the node encoded by bit string γ is on the true path, given q, the probability that the expected outcome from one node to the next is incorrect. Let a and b be the number of `1`'s and `0`'s, respectively. Then,

$$p(\gamma \,|\, q) = p^a q^b.$$

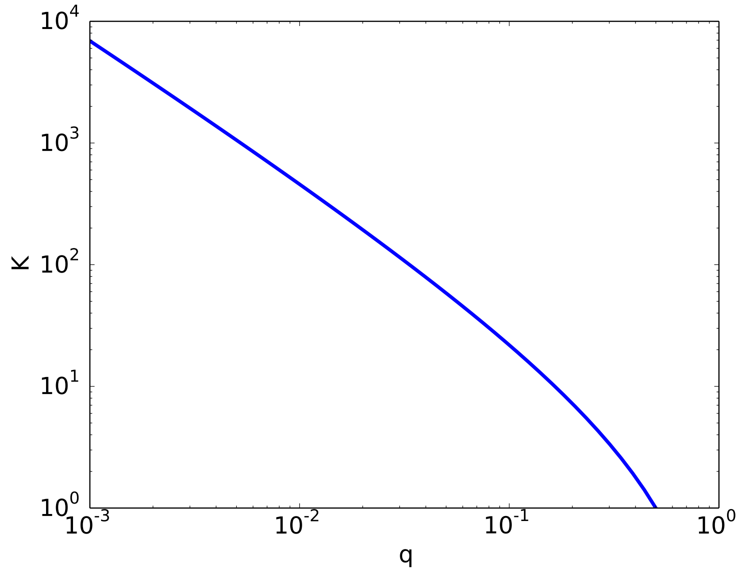


Figure 3.11: K as a function of q, shown on a log-log plot. $K = \log q / \log(1-q)$ is the maximum number of nodes allocated before the greedy scheduling algorithm starts to place cores away from the single true path, and $0 < q < 1/2$ is the error rate in our biased coin model.

The resulting expected speedup is the sum of the J largest terms in

$$U = \{\Pr(\texttt{0}), \Pr(\texttt{1}), \Pr(\texttt{00}), \Pr(\texttt{01}), \Pr(\texttt{10}), \Pr(\texttt{11}), \dots\},$$

where we have suppressed the dependency on q in our notation. Figure 3.12 plots the expected S_J as a function of J, for different values of q; it falls within the bounds mentioned previously. To compute this sum, it is not necessary to exhaustively enumerate all the terms up to depth J; our above reasoning tells us that depth K suffices. Let z be the number of `0`'s, *i.e.*, errors. We consider all terms up to depth K, for $z = 0, \dots, 87$:

- $z = 0 : \Pr(\texttt{1}), \Pr(\texttt{11}), \Pr(\texttt{111}), \dots$
- $z = 1 : \Pr(\texttt{0}), \Pr(\texttt{01}), \Pr(\texttt{10}), \Pr(\texttt{011}), \Pr(\texttt{101}), \Pr(\texttt{110}), \dots$
- $z = 2 : \Pr(\texttt{00}), \Pr(\texttt{001}), \Pr(\texttt{010}), \Pr(\texttt{100}), \dots$
- $\dots$

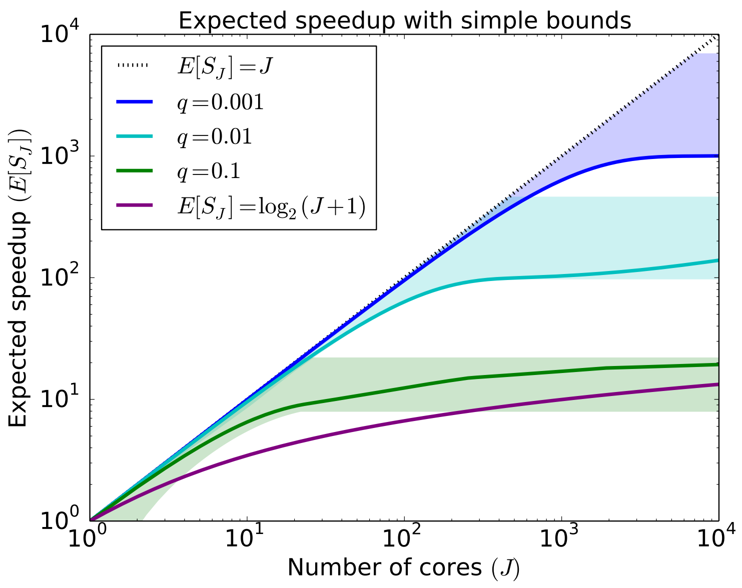
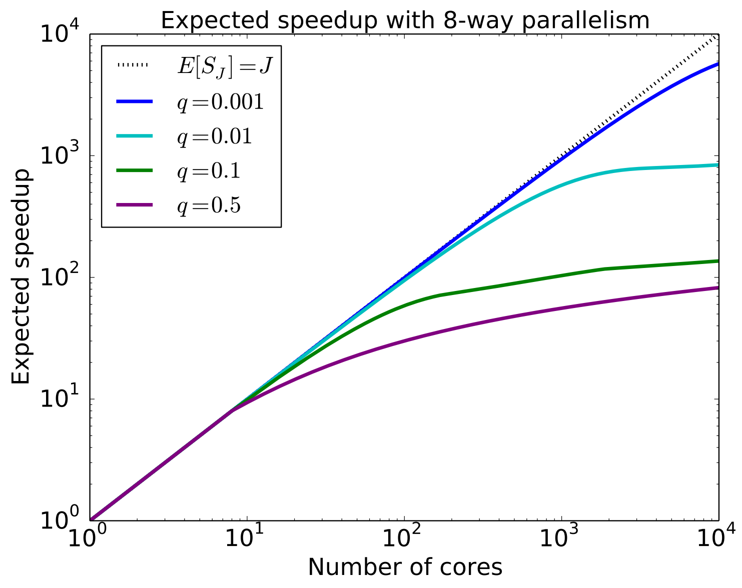


Figure 3.12: Expected speedup as a function of the number of workers. The extremes of the shaded regions are the lower and upper bounds for speedup from Section 3.5.1. The solid lines show the sum of the J largest terms in U, as described in Section 3.5.2. Different colors indicate different values of q. The dotted black line corresponds to perfect linear speedup; the solid purple line corresponds to the naïve scheme, *i.e.*, $q = 0.5$. When $q = 0.1, 0.01, 0.001$, then $K = 21, 458, 6904$, respectively; these values correspond to the (horizontal) upper bounds where $J \geq K$.

We do not actually enumerate all of the above terms, but instead count the number of terms in each group having the same probability, *i.e.*, all terms for a particular z at the same depth in the tree. For example, there are 3 bit strings of length 3 with $z = 1$; they all have probability $p^2 q$.

3.5.3 Speculation plus parallelism at each node

Finally, we consider the case where we combine parallel predictive prefetching with parallelism at each node in the jobtree. Figure 3.13 plots the expected speedup as a function of J, for 8-way and 64-way parallelism at each node. Such a scheme would allow us to place more workers at higher-probability nodes, and therefore achieve greater speedup. As we noted in Section 2.6.2, Strid (2010) has experimented with this idea.



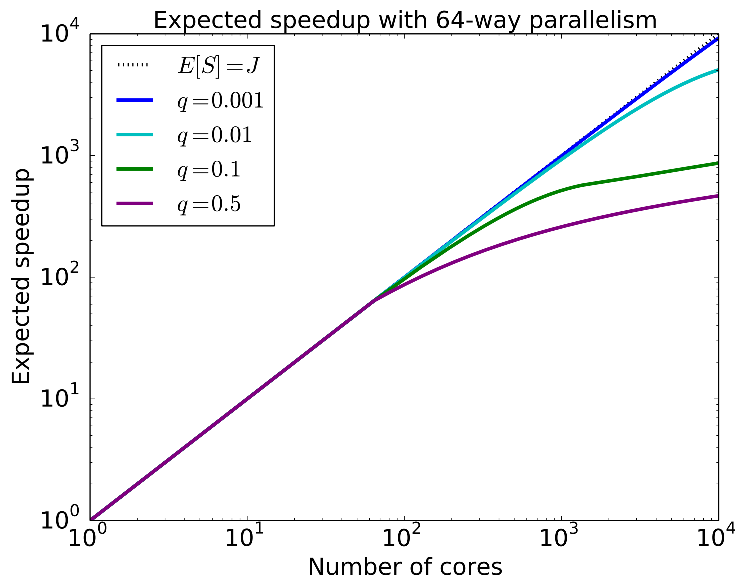


Figure 3.13: Expected speedup as a function of J, with 8-way (top) and 64-way (bottom) parallelism at each node. The dotted black line corresponds to perfect linear speedup; the solid purple line ($q = 0.5$) corresponds to the naïve scheme with parallelism at each node.

Chapter 4

System architecture and implementation

This chapter presents the design and implementation of a practical parallel system for predictive prefetching. For concreteness, we focus on Metropolis–Hastings in the case of large-scale Bayesian inference. First, we give an overview of our system architecture, which follows a master-worker pattern. The master and workers communicate via message passing. The master keeps track of the state of computational work that could be performed, is currently in progress or has been completed by the workers. Importantly, this includes, in the form of probabilities, predictive information about what work is believed to be the most useful. The master uses this information to schedule computational work to be completed by the workers. In the following section, we describe how the master uses the jobtree, introduced in Section 3.2.5, as the central data structure for managing this information. Then, we describe our model of execution as it is driven by the messages passed between the workers and the master. In the next sections, we provide details about how the master manages pseudo-randomness and how the workers generate MH proposals. Finally, we describe our plug-in interface for specifying an instantiation of MH within our predictive prefetching implementation.

4.1 Architectural overview

Our system architecture follows a master-worker pattern requiring $J \geq 2$ parallel cores. One is designated the master and the remaining $J - 1$ are workers. The main components of our architecture are: the protocol by which the master and workers communicate, a data structure for keeping track of computations and their expected utilities, a scheduler that determines what computational work should be performed by each executor, and executors that generate proposals and evaluate the target or approximate posteriors. For clarity, we describe our system in the context of Bayesian posterior sampling. Given a target posterior $\pi(\theta \,|\, \mathbf{x})$, proposal distribution $q(\theta' \,|\, \theta)$, initial state θ_0 and number of iterations T, our system executes a Metropolis–Hastings simulation. The output sequence of samples $\theta_1, \ldots, \theta_T$ is invariant to the number of worker cores. Before describing each architectural component in further detail, we use two state machines to describe the high-level actions of the master and a worker. We also highlight our use of lazy evaluation principles that make our implementation practical.

4.1.1 Master state machine

The central roles of the master are to implement the scheduler that assigns computational work to each worker, cache the workers' computational results and emit the simulated Markov chain. The Markov chain starts in some given initial state. We can describe the actions of the master via a state machine, depicted in Figure 4.1. The master starts in the `wait` state, where it waits for any message from any worker. Eventually, the master receives one of three messages, `WANT-WORK` , `SET-PROPOSAL` or `UPDATE` , from a particular worker. Upon receipt of a `WANT-WORK` message, the master moves to the `schedule` state. There, it identifies useful computational work, replies to the worker with a `HAVE-WORK` message and returns to the `wait` state. Upon receipt of a `SET-PROPOSAL` message, the master moves to the `add-proposal` state. The `SET-PROPOSAL` message contains a proposal computed by the worker, which the master caches before returning to the `wait` state. Upon receipt of an

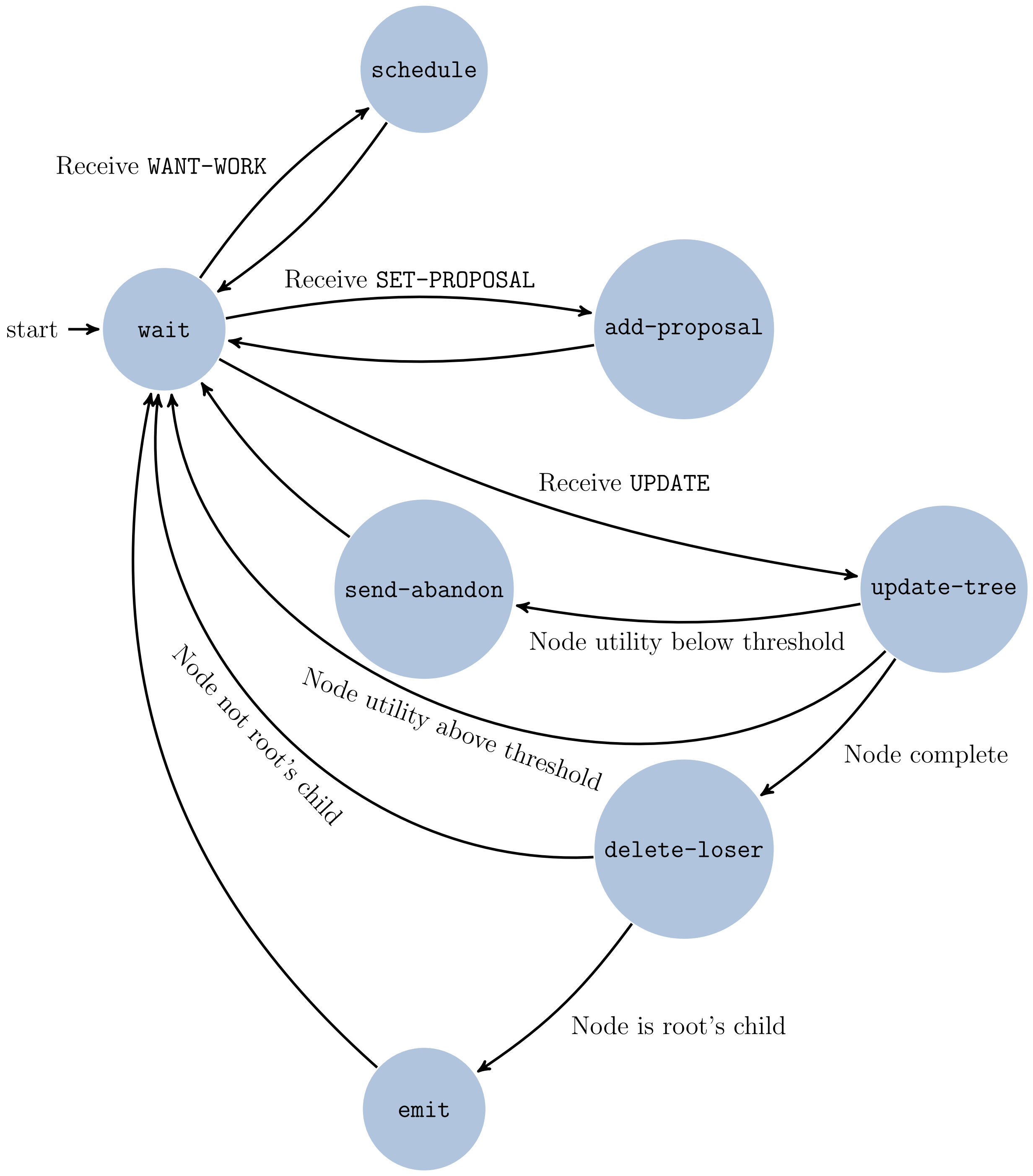


Figure 4.1: State machine for the master.

`UPDATE` message, the master moves to the `update-tree` state. The `UPDATE` message contains information that improves the predictor for some node ρ in the jobtree, as described in Section 3.3. The master uses this information to update the predictor ψ_ρ.

Once the master applies the last update at a node, the node becomes *complete*. If both ρ and ρ's comparison parent are complete, then the predictor ψ_ρ converges to the indicator in Equation 3.3. In this case, one of ρ's children has branch probability 1 and the other has branch probability 0. Consequently, this latter child's entire subtree has utility 0. The master moves to the `delete-loser` state, where it deletes this subtree. If ρ is complete but is not the child of the root in the jobtree, then the master returns to the `wait` state. Otherwise, the master now knows the result at the immediate transition and moves to the `emit` state. There, the master emits one or more of the next states of the Markov chain. The master also integrates garbage collection with updating, and at this point trims portions of the jobtree that are no longer relevant. This could alternately have been integrated with the master's response to some other periodic worker message. Either way, there is no separate garbage collection 'process.' From the `emit` state, the master returns to the `wait` state.

If the update does not contain enough information for the predictor to converge to the indicator, the master may optionally reconsider whether further computation at node ρ is still of interest. If the master decides that the expected utility of ρ is above some threshold, then it returns back to the `wait` state; in this case, the worker continues work on the current node. Otherwise, the master moves to the `send-abandon` state, where it sends the worker an `ABANDON` message, telling the worker to stop its current computation. From the `send-abandon` state, the master returns to the `wait` state.

4.1.2 Worker state machine

The central role of a worker is to implement an executor that performs computational work scheduled by the master. We depict the state machine of a worker in Figure 4.2. The worker starts in the `want` state, in which it sends the master a `WANT-WORK` message indicating that

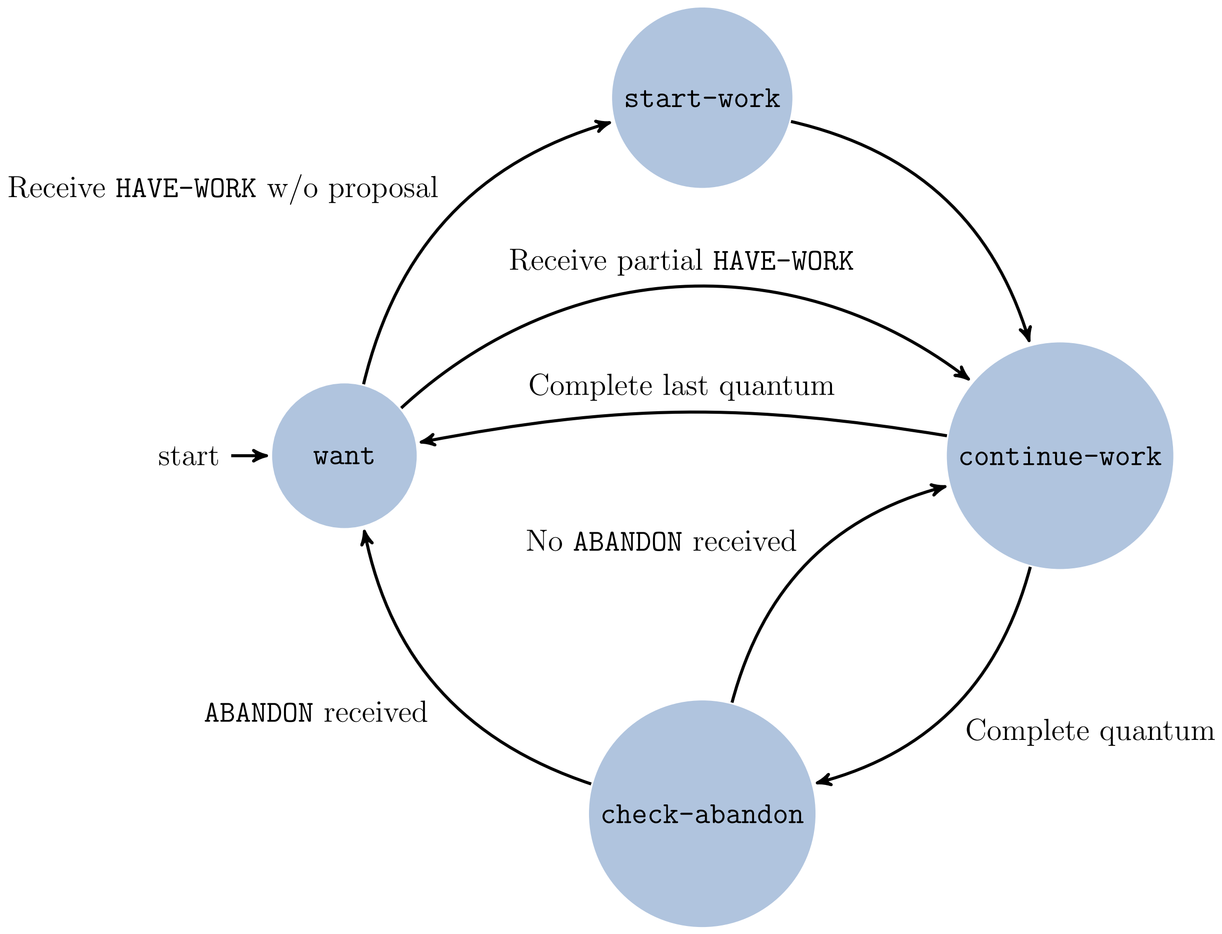


Figure 4.2: State machine for a worker.

it is ready for a new assignment of computational work. The worker leaves the `want` state once it receives a `HAVE-WORK` message from the master containing such an assignment.

If the worker receives a `HAVE-WORK` message for a node ρ in the jobtree such that the message does not contain the state θ_ρ, then the worker moves to the `start-work` state. Note that the first `HAVE-WORK` message for the root of the jobtree contains the initial state θ_0, while the first `HAVE-WORK` message for any other node does not contain the corresponding state, which in this latter case is always a proposal. In the `start-work` state, the worker generates the proposal θ_ρ at node ρ and can compute functions of θ_ρ, *e.g.*, in Bayesian posterior sampling, the prior $\pi_0(\theta_\rho)$ in Equation 3.7 is only a function of θ_ρ. The worker sends these results to the master and then moves to the `continue-work` state. The worker could alternately receive a work assignment to resume computation at a partially evaluated node; in this case, it moves directly from the `want` state to the `continue-work` state.

In the `continue-work` state, the worker performs computation that contributes to forming a predictor ψ_ρ for the given node ρ. For example, it may compute a fast approximation to the target likelihood evaluated at θ_ρ, which together with the prior produces an approximate posterior. Alternatively, it may complete the exact target evaluation of interest, or any incremental portion of any approximate target or the exact target. After computing some *quantum*, *i.e.*, useful unit of computation, the worker moves to the `send-update` state, where it sends a message to the master containing the latest results produced while in the `continue-work` state. If at this point the worker has completed the last assigned quantum at node ρ, then it moves back to the `want` state. Otherwise, the worker enters the `check-abandon` state, where it checks whether it has received an `ABANDON` message from the master instructing it to stop its current computation. If it has received such a message, then it returns to the `want` state; otherwise, it returns to the `continue-work` state.

4.1.3 Practical considerations

Lazy evaluation is an important principle that makes our implementation practical. In particular, our central data structure is a binary tree with depth equal to the number of desired Metropolis–Hastings samples, *i.e.*, the number of iterations in the equivalent serial execution; in our empirical studies designed to be representative of realistic scenarios, this number is 50000. We only materialize small portions of the tree as they become useful for prefetching; we never materialize the whole tree. Furthermore, the expected utilities change as computations are updated and completed, so we lazily compute expected utilities only as they are needed, instead of continuously updating them.

4.2 The jobtree

Central to our architecture is a data structure for keeping track of computations and other information relevant to managing a MCMC simulation within the predictive prefetching framework. This data structure is used for caching the results of computation performed by the workers, including speculative and partial computation. Furthermore, it must be able to maintain and update probabilistic beliefs about the utilities of different computations that have been or could be performed or are currently in progress. During the course of execution, this data structure must reflect when these beliefs eventually converge to certainty and thus support the construction of output exactly equal to the equivalent serial computation. Importantly, it must be efficient to query this data structure for computational work that is not complete, is not currently being performed by any worker and is believed to be significantly useful.

A tree supports our data structure requirements. Specifically, the master maintains a representation of the jobtree, introduced in Section 3.2.5. Every node ρ in the jobtree is associated with a state θ_ρ and full, partial or approximate computation of the target density at that state. If the full target density $\pi(\theta_\rho \,|\, \mathbf{x})$ has been computed, then we mark the node as *complete*; otherwise, the partial or approximate computation provides an estimate $\tilde{\pi}(\theta_\rho \,|\, \mathbf{x})$ and we mark the node as *incomplete*. Every node ρ, except the root, has a right child $\rho\mathtt{1}$, a left child $\text{flip}(\rho)\mathtt{1}$ and a comparison parent $\lfloor \text{flip}(\rho) \rfloor$. The root ϵ has a single (right) child $\epsilon\mathtt{1}$. Every node ρ, except for the root, is a proposal and has a predictor ψ_ρ equal to the conditional probability that θ_ρ is accepted, given that ρ is on the true computation path. For convenience, we set the root's predictor to 1. Every node also has an expected utility, *i.e.*, probability of being on the true computation path. The root's (expected) utility is 1. For every other node ρ, the expected utility is the product of the branch probabilities along the path connecting the root to ρ. Recall that the branch probabilities label the edges and are related to the predictors: the edge from a node ρ to its right child has branch probability equal to the predictor ψ_ρ and the edge to its left child has branch probability $1 - \psi_\rho$.

Our jobtree includes several features specific to our implementation. As described in Section 4.1.3, the master lazily computes the expected utility of a node ρ whenever called for by traversing the path from the root to ρ. Thus, the expected utilities are not explicitly represented at each node in the jobtree. Also, every node ρ may have at most one *executor*, *i.e.*, the worker core computing $\pi(\theta_\rho \,|\, \mathbf{x})$, and optional executor status information, such as how much partial computation has been completed so far. An extension to our implementation would be to have multiple executors per node, an idea explored by Strid (2010) and by us in Section 3.5.3, and that we discuss further in Chapter 6. Finally, every node is marked with one of three designations. A *dead* node has utility 0; it cannot be part of the true computation path. A *pending* node is incomplete, has positive expected utility and has no executor. An *active* node is incomplete, has positive expected utility, and has an executor.

4.3 Selecting high-utility pending nodes

One of the master's jobs is to assign workers to the pending nodes with highest utility. In our initial implementation, the master maintained a priority queue called the *pending queue*. It contained all pending nodes in the tree, ordered by *ascending* expected utility, *i.e.*, the head of the pending queue was the inactive node with the highest expected utility.

We observed that the pending queue was actually redundant with the jobtree, as the latter data structure contained all the same information. This suggested an alternative stochastic routine, described next, that allowed us to eliminate the pending queue. To assign a worker to a node, the master stochastically traverses the jobtree from the root, following branches according to their branch probabilities, until it finds a pending node, *i.e.*, a node that is neither active nor dead and has no executor. In this way, the master stochastically assigns workers to those nodes with highest expected utility.

4.4 Execution and messaging protocol

Our system's execution is driven by the messages communicated from the workers to the master and vice versa. The protocol by which they communicate is implicit in their state machines, described in Section 4.1. We present our model of execution and messaging protocol together by considering the various possible scenarios that may occur.

Initially, the jobtree contains two nodes, ϵ and `1`. Both of these nodes have (expected) utility 1. The predictor $\psi_{\mathtt{1}}$ is initialized to 0.5, reflecting that we initially have no information about whether the first proposal will be accepted or rejected. The master can send or receive messages to any worker and each worker can send or receive messages to the master; the workers do not communicate with each other. From the master's state diagram in Figure 4.1, we can see that its actions are responsive, *i.e.*, it starts in the `wait` state, leaves only to respond to a message from a worker, and always returns to the `wait` state.

Workers initiate execution by requesting work from the master via a `WANT-WORK` message. This is the first action of each worker. We summarize the simplest initial series of messages, starting with such a `WANT-WORK` message, in Figure 4.3. When the master receives a `WANT-WORK` message from worker W, it finds a pending node with high expected utility, as described in Section 4.3. The master replies to worker W with a `HAVE-WORK` message containing ρ and, if known, the state θ_ρ, and otherwise, it also contains whatever information is available to generate θ_ρ most efficiently. Precisely what information this is will become clear in Sections 4.5 and 4.6; for now, note that θ_ρ can be generated from its comparison parent, $\theta_{\lfloor \mathrm{flip}(\rho) \rfloor}$, given the appropriate position in the pseudorandom stream. Further, recall that the log (unnormalized) target posterior decomposes as

$$\mathcal{L}(\theta_\rho) = \log \pi(\theta \,|\, \mathbf{x}) = \log \pi_0(\theta) + \sum_{n=0}^{N-1} \log \pi(x_n \,|\, \theta). \tag{4.1}$$

In our implementation, workers compute the above sum of likelihood terms in batches of size b, and thus it may have already been partially evaluated. Also in batches, workers

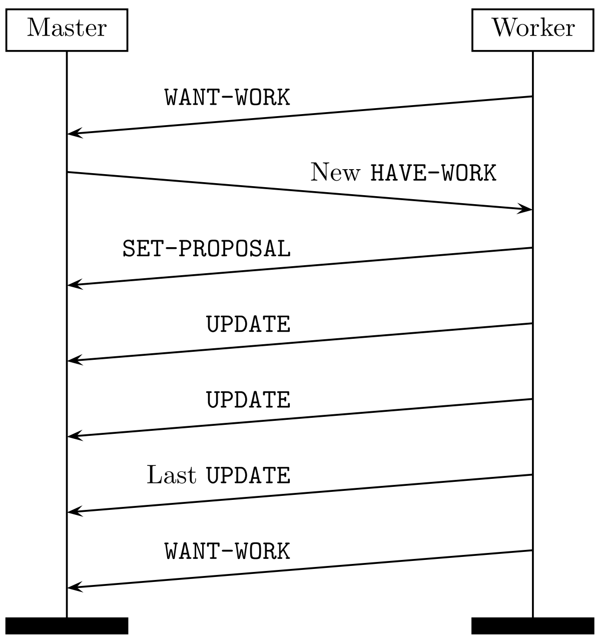


Figure 4.3: Simplest scenario in which a worker asks for work via a `WANT-WORK` message and receives a new `HAVE-WORK` message for a particular node in the jobtree. The worker performs all the work at this node, sending partial results along the way. The proposal, and potentially, initial computational results involving the proposal, are contained in a single `SET-PROPOSAL` message. Subsequent computational results are contained in one or more `UPDATE` messages. When the worker is finished, it requests new work with via another `WANT-WORK` message. For simplicity, this diagram contains only one worker.

compute the sum of squared likelihood terms

$$\sum_{n=0}^{N-1} (\log \pi(x_n \,|\, \theta))^2 \tag{4.2}$$

used by the master to estimate the error in using a partially evaluated target to approximate the true target. Thus, the `HAVE-WORK` message also includes the results of having partially evaluated Equations 4.1 and 4.2, as well as an index m, where $0 \leq m < N-1$ and $m \mod b = 0$, indicating where the worker should resume computation. The master then marks ρ as active and sets ρ's executor to W. If ρ does not yet have any children in the jobtree, the master also creates ρ's children, each of which it marks as pending. For each new node ϕ, its predictor ψ_ϕ is initialized to the *local acceptance rate*, which we define to be empirical acceptance rate observed during simulation of the k most recent Metropolis–Hastings samples. In our implementation, $k = \min\{t, 100\}$, where t is total number of MH samples obtained thus far.

A HAVE-WORK message for node ρ contains the partially evaluated target posterior in Equation 4.1, the partially evaluated sum of squared likelihood terms in Equation 4.2, an index m and, if $m > 0$, the state θ_ρ. Figure 4.3 depicts the scenario where the HAVE-WORK message does not contain θ_ρ. In this case, the worker first generates θ_ρ, as described in Section 4.6, and then computes the log prior $\log \pi_0(\theta_\rho)$. The worker replies to the master with a SET-PROPOSAL message containing ρ, θ_ρ and $\log \pi_0(\theta_\rho)$. The master stores the information from this message in the jobtree, *i.e.*, it stores θ_ρ and, if relevant, $\log \pi_0(\theta_\rho)$ at node ρ.

Alternately, the HAVE-WORK message already contains the proposal, and potentially, additional partial results; this scenario is depicted in Figure 4.4. In either case, the worker proceeds with computing the target posterior in Equation 4.1 and sum of squared likelihood terms in Equation 4.2, in batches of size b starting at index m. After each completed batch, the worker sends an UPDATE message to the master containing the updated values of the partially evaluated target posterior and sum of squared likelihood terms, as well as an index $m' > m$, indicating how far these incremental computations have progressed in total. The worker periodically checks for any ABANDON messages from the master. Upon receiving such a message, the worker discontinues work on node ρ and sends the master a WANT-WORK message. We depict this scenario in Figure 4.5. In our implementation, the worker makes these checks after each completed batch. After the worker sends the last UPDATE message for the last batch, it sends the master a WANT-WORK message.

Now suppose the master receives an UPDATE message for node ρ from a worker; we depict such messages in Figures 4.3, 4.4 and 4.5. At node ρ in the jobtree, the master updates its estimate of the log posterior as in Equation 4.1 and the error of this estimate as in Equation 4.2. Next, the master updates the branch probabilities, introduced in Section 3.3, of node ρ and any nodes for which ρ is the comparison parent. Recall that each Metropolis–Hastings transition stochastically chooses between two states – where one is the comparison

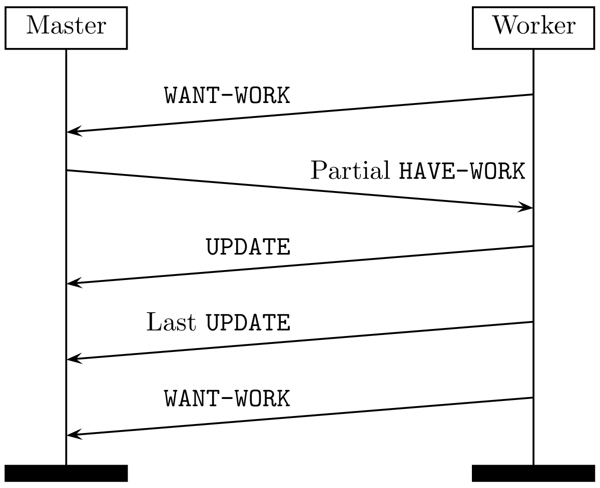


Figure 4.4: Scenario in which a worker receives a partial HAVE-WORK message for node ρ that already contains the state θ_ρ, and potentially, additional partial results.

parent of the other – by comparing their posterior evaluations relative to a uniform random variate. Consider one of these branch probability updates, such that ρ is the comparison parent of a node corresponding to the proposal θ'_ρ with uniform variate u. Recall from Section 3.3 that the edge from a node ρ to its right child has branch probability equal to the predictor ψ_ρ and the edge to its left child has branch probability $1 - \psi_\rho$. To update the predictor ψ_ρ in Equation 3.16, the master computes an estimate of the difference of log posteriors as in Equation 3.11,

$$\mathcal{L}(\theta_\rho) - \mathcal{L}(\theta_\phi) = \log \pi(\mathbf{x} \,|\, \theta_\rho) - \log \pi(\mathbf{x} \,|\, \theta_\phi),$$

the error of this estimate corresponding to Equation 3.14 and the constant value

$$r = u \frac{q(\theta'_\rho \,|\, \theta_\rho)}{q(\theta_\rho \,|\, \theta'_\rho)}.$$

We elaborate on the details of this calculation in Section 4.7. In all of our experiments, the proposal distribution $q(\cdot \,|\, \cdot)$ is symmetric, in which case $r = u$.

When the master applies the last update at node ρ, the node becomes complete. If both ρ and ρ's comparison parent are complete, then the predictor ψ_ρ converges to the indicator in

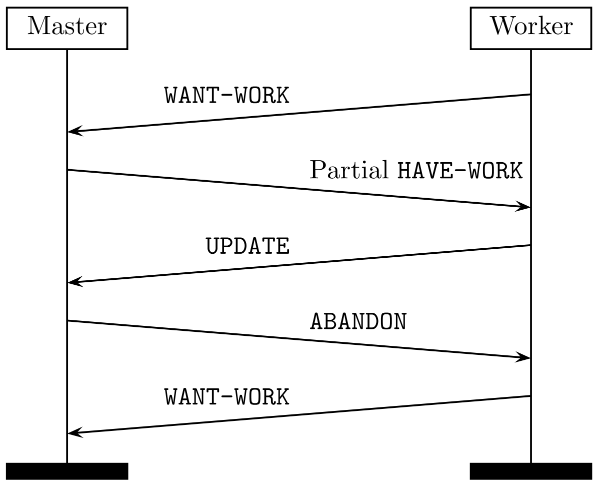


Figure 4.5: Scenario in which a worker receives an `ABANDON` message. If an `UPDATE` from a worker results in the expected utility of the node dropping below some threshold, then the master sends an `ABANDON` message.

Equation 3.3 and the master deletes the subtree with utility 0. If ρ is the child of the root in the jobtree, *i.e.*, the immediate transition chooses between the states at these two nodes, then the master emits the next state of the Markov chain, which is now completely specified, and also emits any subsequent states of the Markov chain whose predictors have already converged to the indicator. Each time the next state of the Markov chain is emitted, the root of the jobtree is removed and the emitted state becomes the new root. If the emitted state corresponds to an accepted proposal, the root's (only) child becomes the new root and the left subtree is trimmed away. Otherwise, the proposal was rejected and old root is still the new root, but needs to be connected to the left subtree at its left grandchild (with bit string `01`) and the right subtree is trimmed away.

If the update does not contain enough information for the predictor to converge to the indicator, the scheduler may optionally reconsider whether further computation at node ρ is still of interest. Recall that each updated branch probability may change the expected utilities of descendant nodes that are pending. The master lazily updates the expected utility of ρ and identifies a high-utility pending node ϕ following the procedures outlined in Sections 4.1.3 and 4.3. If the expected utility of ϕ is significantly greater than that of ρ, the master marks ρ

as pending and sends the worker an `ABANDON` message, depicted in Figure 4.5. Note that any partial computations at an abandoned node remain cached on the jobtree until the node is trimmed. Such abandoned nodes can later be reassigned to workers if their expected utility increases relative to the active nodes. Any subsequent workers will resume computation where the previous worker left off.

In all of our experiments, the master decides to abandon ρ if the expected utility of ϕ is at least a factor of 1.1 greater than that of ρ. This threshold, which we observed empirically to be effective, balances the ideal policy – keeping the workers active at those nodes with highest expected utility – with the actual implementation costs of reassigning nodes to workers, which include some bookkeeping and communication.

4.5 Managing pseudo-randomness

Prefetching schemes require careful management of pseudo-randomness to yield output that is invariant to the number of worker cores, and, as a consequence, is exactly equal to an equivalent serial execution. In Section 3.2.4, we outlined several strategies for achieving this. Here, we describe our approach, in which our system synchronizes the use of pseudo-randomness to our notion of ground truth corresponding to serial execution.

Our implementation follows directly from the mathematical framework presented in Section 3.1. Recall that progressing from one level to the next in the MH binary tree, or equivalently the jobtree, corresponds to one application of the MH transition operator. For a large class of MCMC algorithms, we showed how to decompose the transition operator $T : \mathcal{X} \times \mathcal{U} \to \mathcal{X}$ into two functions. The first function $Q : \mathcal{X} \times \mathcal{U}_Q \to \mathcal{P}(\mathcal{X})$, produces a countable set of candidate points in $\mathcal{X}$, where $\mathcal{P}(\mathcal{X})$ is the power set of $\mathcal{X}$. The second function $R : \mathcal{P}(\mathcal{X}) \times \mathcal{U}_R \to \mathcal{X}$ then chooses one of the candidates for the next state in the Markov chain. $\mathcal{U}_Q$ and $\mathcal{U}_R$ indicate disjoint subspaces of the unit hypercube $\mathcal{U}$ relevant to each part of the operator. In Metropolis–Hastings, $\mathcal{U}_Q$ corresponds to the pseudo-random

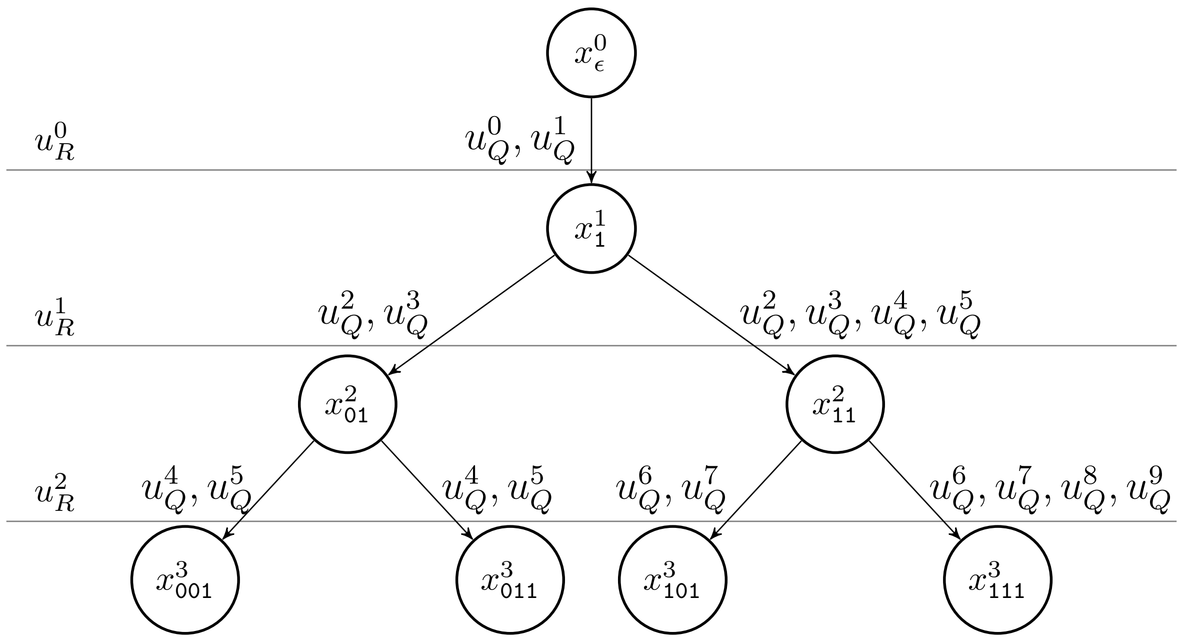


Figure 4.6: Consumption of pseudo-randomness with respect to the Metropolis–Hastings jobtree. Edges are labeled by elements from the pseudo-random sequence $\{u_Q^0, u_Q^1, u_Q^2, \dots\}$ consumed during proposal generation. Each layer also requires one element of a separate pseudo-random sequence $\{u_R^0, u_R^1, u_R^2, \dots\}$, shown on the left, corresponding to the uniform random variate used in the stochastic accept/reject decision. In general, each proposal generation may require a variable number of sequence elements.

numbers consumed to generate proposals and $\mathcal{U}_R$ corresponds to the uniform random variates used to stochastically accept or reject proposals.

In our implementation of MH within the prefetching framework, the proposals are generated on the workers and the decision to accept or reject a proposal is made on the master. We use two pseudo-random sequences: one on the master for the uniform variates used in the stochastic decisions and one shared across the workers for proposal generation. In Figure 4.6, we illustrate the consumption of both sequences with respect to the MH jobtree. The pseudo-random sequence $\mathbf{u}_R = \{u_R^0, u_R^1, u_R^2, \dots\}$ on the master is easy to manage; each MH iteration consumes exactly one element in this sequence. Specifically, all possible transitions that decide between a node at depth d and its comparison parent consume the same element u_R^d, where the depths are indexed starting from 0 at the root. The second pseudo-random sequence $\mathbf{u}_Q = \{u_Q^0, u_Q^1, u_Q^2, \dots\}$ is managed by the master and consumed by the workers; all workers have access to the sequence. As discussed in Section 3.2.4, each proposal generation can require a variable amount of pseudo-randomness, resulting in path-dependent consumption of this sequence. Our master synchronizes the consumption of $\mathbf{u}_Q$ to an equivalent serial evaluation by using the jobtree to keep track of the sequence position before and after each proposal is generated.

Note that we could have instead used a single pseudo-random sequences; this would correspond most closely with standard implementations of serial MH and would have effectively interleaved our two sequences. In this scenario, the uniform variates are path-dependent rather than simply a function of MH (job)tree depth, or equivalently, iteration. Maintaining two sequences allows all the uniform variates to be specified at the beginning of computation, which enables their immediate use by predictors.

4.6 Generating proposals

In our implementation, the executors on the workers compute the proposals; an alternative design could have the master compute them. Our focus is on the regime where proposal generation is fast relative to target evaluation, thus either choice is reasonable. Given a state θ and proposal distribution $q(\cdot \,|\, \cdot)$, a proposal $\theta' \sim q(\theta' \,|\, \theta)$ is generated by sampling from the distribution. A common choice is to sample according to a symmetric distribution, *e.g.*, a Gaussian distribution centered at θ, as in $\theta' \sim \mathcal{N}(\theta' \,|\, \theta, \sigma^2)$.

The master and workers communicate to synchronize consumption of $\mathbf{u}_Q$ across the workers. Each proposal θ' depends directly on its comparison parent θ as well as some pseudo-randomness via the sequence $\mathbf{u}_Q$. As explained in Section 4.5, the consumption of this sequence is, in general, path-dependent with respect to the Metropolis–Hastings jobtree. When the master sends a worker a `HAVE-WORK` message indicating that the worker needs to generate a proposal, this message also contains an index indicating the last used position j in the $\mathbf{u}_Q$ sequence. The worker generates the proposal, consuming k sequence elements $\{u_Q^{j+1}, u_Q^{j+2}, \ldots, u_Q^{j+k}\}$. When the worker responds with a `SET-PROPOSAL` message, it includes the proposal as well as the index $j + k$, which the master records on the jobtree. This allows the master to keep track of the information contained in Figure 4.6.

Notice that a proposal cannot be generated until all its ancestors in the jobtree have already been generated, regardless of whether they correspond to accepted or rejected proposals. The simplest approach would be for the master to assign a worker to a node only after the proposal at its parent is known, *i.e.*, after the proposals at all its ancestors have been generated. This leads to a startup problem, where potentially many workers are available but cannot be assigned immediately to nodes. Our solution is to enable workers to generate the proposal θ_ρ at node ρ from the state at its comparison parent, or if that is not available, the comparison parent of its comparison parent, or any such *ancestral comparison parent* α further back in the jobtree. To accomplish this, the master transmits a `HAVE-WORK` message containing the state θ_α at the ancestral comparison parent α closest to ρ, the index of the last element in $\mathbf{u}_Q$ used to generate θ_α and an encoding of the path on the jobtree from α to ρ. This code is a string indicating the sequence of right and left branches on the path. Given this information, the worker simulates the corresponding sequence of accept and reject decisions, generating each proposal on the path until it produces the desired state. The worker then transmits a `SET-PROPOSAL` message containing this state and the index of the last consumed element of $\mathbf{u}_Q$.

4.7 Predictor implementation

The target posteriors $\log \pi(\theta \,|\, \mathbf{x})$ and $\log \pi(\theta' \,|\, \mathbf{x})$ are evaluated by separate workers, as described in Section 4.4. Our normal model for the Metropolis–Hastings ratio based on a subsample of size m, derived in Section 3.4, depends on the empirical mean and standard deviation of the differences Δ_n from Equation 3.11. We use an approximation to our error model that avoids having to keep track of all these differences, since this would require extra communication. The worker for θ calculates

$$G_m(\theta) = \log \pi_0(\theta) + \frac{N}{m} \sum_{n=1}^{m} \log \pi(x_n \,|\, \theta) \tag{4.3}$$

rather than the difference mean $\hat{\mu}_m$ from Equation 3.13. Given these values, the master can precisely compute $\hat{\mu}_m = G_m(\theta') - G_m(\theta)$, but the empirical standard deviation of differences, s_m in Equation 3.14, must be estimated. Recall that for two random variables X and Y with standard deviations σ_X and σ_Y, respectively, and covariance $\sigma_{X,Y}$, the standard deviation of their difference is $\sqrt{\sigma_X^2 + \sigma_Y^2 - 2\sigma_{X,Y}}$. Also, their covariance is related to their correlation $\rho_{X,Y}$ and standard deviations via $\sigma_{X,Y} = \rho_{X,Y}\sigma_X\sigma_Y$. Treating the likelihood terms as random variables and combining the above two facts gives

$$s_m = \sqrt{S_m(\theta)^2 + S_m(\theta')^2 - 2\tilde{c} S_m(\theta) S_m(\theta')}\,, \tag{4.4}$$

where $S_m(\theta)$ denotes the empirical standard deviation of the first m terms $\log \pi(x_n \,|\, \theta)$, and $\tilde{c}$ approximates the correlation between $\log \pi(x_n \,|\, \theta)$ and $\log \pi(x_n \,|\, \theta')$. We empirically observe this correlation to be very high; in all experiments we set $\tilde{c} = 0.9999$. Note that this approximation affects only the quality of our speculative predictions; it does not affect the actual decision to accept or reject the proposal θ'.

4.8 Implementation details and plug-in interface

Our implementation is written primarily in C++.[1] We make use of several Boost C++ libraries, including the Boost.MPI implementation of the Message Passing Interface (MPI) for communication between the master and worker cores, its Serialization library for constructing messages and Boost.Random for the Mersenne twister pseudo-random number generator used by $\mathbf{u}_R$ and $\mathbf{u}_Q$.

[1]Code for our implementation is publicly available at `https://github.com/elaine84/fetching`.

Algorithm 4 Our two-core implementation of Metropolis–Hastings for Bayesian posterior sampling with a symmetric proposal distribution. Messages are suppressed for simplicity.

```
Input: Initial state $\theta^0$, number of iterations $T$, data $\mathbf{x}$, log prior $\log \pi_0(\theta)$, log likelihood $\log \pi(\theta \mid x_n)$, proposal function $\theta' \sim q(\theta' \mid \theta)$
Output: Samples $\theta^1, \dots, \theta^T$
$\log \pi(\theta^0 \mid \mathbf{x}) = \log \pi_0(\theta^0) + \sum_{n=1}^{N} \log \pi(x_n \mid \theta^0)$    ▷ Worker evaluates target at $\theta^0$
for $t$ in $0, \dots, T-1$ do
    $\theta' \sim q(\theta' \mid \theta)$    ▷ Worker generates proposal, which consumes $\mathbf{u}_Q$
    $\log \pi(\theta' \mid \mathbf{x}) = \log \pi_0(\theta') + \sum_{n=1}^{N} \log \pi(x_n \mid \theta')$    ▷ Worker evaluates target at $\theta'$
    $u_R^t \sim \text{Unif}(0,1)$    ▷ Master samples uniform random variate, which consumes $\mathbf{u}_R$
    $\theta^{t+1} \leftarrow \begin{cases} \theta' & \text{if } \log u_R^t < \log \pi(\theta' \mid \mathbf{x}) - \log \pi(\theta^t \mid \mathbf{x}) \\ \theta^t & \text{otherwise} \end{cases}$    ▷ Master selects next state $\theta^{t+1}$
end for
```

Algorithm 4 presents a summary of our implementation of MH for Bayesian posterior sampling, with two cores. As can be seen from this description, an instantiation of our system depends on user-defined functions for the log prior $\log \pi_0(\theta)$, log likelihood $\log \pi(\theta \mid \mathbf{x})$, and proposal distribution $\theta' \sim q(\theta' \mid \theta)$. Given a fixed dataset $\mathbf{x}$, the log likelihood is evaluated in batches, therefore this function also takes as arguments an index into $\mathbf{x}$ and batch size b. All our experiments use symmetric proposal distributions, thus do not require a user-defined log proposal density $\log q(\theta' \mid \theta)$.

Our system includes a plug-in interface for these user-defined functions and supports user functions callable from C++, Python or the command-line. The command-line interface allows us to support user functions written in other popular languages for scientific computing, such as MATLAB and R. These functions are called by executors implemented in C++. Note that the user-defined proposal function $\theta' \sim q(\theta' \mid \theta)$ must use the pseudo-random stream $\mathbf{u}_Q$ under the control of the master. For a pure C++ instantiation of our system, it is straightforward to synchronize the direct use of $\mathbf{u}_Q$ across workers. For proposal functions written in other languages, the simplest approach is for the master to supply appropriate access to $\mathbf{u}_Q$ that the proposal function can use to seed some native pseudo-random number generator. In particular, Python user functions can access $\mathbf{u}_Q$ via a `rand` module that we provide and includes familiar function interfaces, *e.g.*, `rand.random()` for the next random float in $[0.0, 1.0)$.

We focus on describing our plug-in interface for user-defined functions written in Python because it is a language widely used among machine learning practitioners. In the experiments presented in the next chapter, all user functions are written in Python. An executor calls Python user functions via the Boost.Python library. The user specifies these functions as methods of a single class. Note that in their descriptions below, θ is a Python object.

- `__init__` is the first method called by the executor to perform one-off, initialization actions such as loading the dataset $\mathbf{x}$.
- `data_size` returns N, the number of elements in $\mathbf{x}$, used by the executor to track the progress of evaluating the log likelihood.
- `first_proposal` returns the initial state θ^0, which it might generate randomly.
- `next_proposal` takes as input θ, calls a function from our `rand` module, described above, and uses the returned value to seed a Python pseudo-random number generator that it uses to generate a proposal $\theta' \sim q(\theta' \,|\, \theta)$.
- `log_prior` takes as input θ and returns the log prior, $\log \pi_0(\theta)$.
- `evaluate` takes as input θ and returns a batch of work towards evaluating the log likelihood, $\sum_{n=1}^{N} \log \pi(x_n \,|\, \theta)$.
- `unparse_proposal` takes as input θ and returns a string representation that the executor serializes when constructing a `SET-PROPOSAL` message. Notice that the master never needs to interpret this representation, it simply caches it on the jobtree.

Our user-defined Python functions make use of the NumPy package for its pseudo-random number generator, convenient random sampling functions (*e.g.*, for generating proposals sampled from Gaussian distributions) and array operations (*e.g.*, for the batched log likelihood evaluations).

Chapter 5

Empirical evaluation

Our evaluation focuses on Metropolis–Hastings for large-scale Bayesian inference, using the predictors described in Sections 3.4 and 4.7, though our framework can use any approximation scheme for the target distribution. Our experiments rely on realistic modeling problems of interest to the machine learning community. Furthermore, we design our experiments to be representative of typical MH simulation in practice. Specifically, we design each experiment to start away from convergence, progress through burn-in and eventually converge, according to standard statistics, while achieving a reasonable acceptance rate and number of effective samples. In this chapter, we first describe two Bayesian inference problems that we selected as nontrivial benchmarks – the first uses synthetic data and the second uses real data. One challenge for our evaluation was to design benchmarks representative of realistic scenarios involving MH sampling. This led to our adoption of an adaptive MH algorithm, which we implemented as a small extension to our original framework for standard MH. We justify and describe this algorithm and its implementation. We also provide a thorough explanation of how we assess chain convergence, use this framework to provide a definition of burn-in, and assess the quality of samples obtained after convergence. Next, we evaluate our system's implementation, described in Chapter 4, with up to 64 worker cores in a multicore cluster environment. We report our main speedup results relative to serial computation in our system, *i.e.*, with one master and one worker. We then characterize the behavior of the adaptive MH algorithm over the course of execution. This in turn helps us understand the behavior of our predictor, which is the primary determinant behind our speedup results. To understand our implementation's inefficiencies, we present further measurements of our system's performance that decouple the effect of inaccurate predictions from other system overheads. We conclude with a discussion of these overheads and suggest methods for addressing them.

5.1 Example Bayesian inference problems

We evaluate our system on both synthetic and real Bayesian inference problems. For each problem, the posterior is a standard but interesting probabilistic model that is described by a multidimensional parameter vector ($d > 50$) and whose likelihood is a function of a large dataset ($N \geq 10^6$). During the development of our system, we employed several other problems as benchmarks, but do not include them in our evaluation here because they relied on either synthetic data generated from a simple posterior model or real datasets with relatively small numbers of data points.

5.1.1 Mixture of multidimensional Gaussians

Our first target distribution is the posterior density of the eight-component mixture of eight-dimensional Gaussians used by Nishihara et al. (2014), where the likelihood is a function of $N = 10^6$ samples drawn from this model. The data is thus described by a matrix $X \in \mathbb{R}^{N \times d}$, where $d = 8$. The posterior density over the model parameters θ is

$$\pi(\theta \mid X) \propto \pi_0(\theta)\pi(X \mid \theta).$$

We use a uniform prior, $\pi_0(\theta) \propto 1$, and the likelihood function is

$$\pi(X \mid \theta) = \sum_{k=1}^{8} w_d \, \mathcal{N}(X \mid \mu_k, 1) \propto \sum_{k=1}^{8} w_k \prod_{n=1}^{N} e^{-\frac{1}{2}(x_n - \mu_k)^\top (x_n - \mu_k)}.$$

We use equal mixture weights, setting $w_k = 1$ and place the means at $\mu_k = \ell \phi_k - \ell/2$, where $\ell = 4$ and every component of each ϕ_k is drawn uniformly at random from the interval $[0, 1)$. See Appendix A for the ϕ_k values used in our experiments. The parameter vector θ concatenates the means μ_k, thus is 64-dimensional and real-valued.

5.1.2 Bayesian Lasso for photovoltaic activity

Our second target distribution is the posterior density of a Bayesian Lasso (least absolute shrinkage and selection operator) regression that models molecular photovoltaic activity. The likelihood involves a dataset of $N = 1.8 \times 10^6$ molecules described by 56-dimensional real-valued cheminformatic features (Olivares-Amaya et al., 2011; Amador-Bedolla et al., 2013); each response is real-valued and corresponds to a lengthy density functional theory calculation (Hachmann et al., 2011, 2014).[1] Thus, the data is described by a matrix $X \in \mathbb{R}^{N \times d}$, where $d = 56$, and the responses are a (column) vector $\mathbf{y} \in \mathbb{R}^N$.

The Lasso is a linear regression method that penalizes the absolute values of the regression coefficients through an ℓ_1 penalty (Tibshirani, 1994). Assuming mean-centered data $X = \mathbf{x}_1, \dots, \mathbf{x}_n$, linear regression models the response data $\mathbf{y} = y_1, \dots, y_n$ according to $y_n \sim \mathcal{N}(\mathbf{x}_n^\top \beta, \sigma^2)$, where $\beta \in \mathbb{R}^d$. Ordinary least squares solves for the coefficient vector β that minimizes the sum of squared residuals,

$$\min_{\beta} \sum_{n=1}^{N} (y_n - \mathbf{x}_n^\top \beta)^2 = \min_{\beta} \, (\mathbf{y} - X\beta)^\top (\mathbf{y} - X\beta).$$

[1]For the specific features used here, we thank Michael Tingley.

Lasso adds an ℓ_1 penalty on β,

$$\min_{\beta}\; (\mathbf{y} - X\beta)^\top(\mathbf{y} - X\beta) + \lambda|\beta|_1,$$

where $|\beta|_1 = \sum_{i=1}^{d} |\beta_i|$, for some $\lambda \geq 0$. This penalty has the effect of encouraging β to be sparse. Park and Casella (2008) take a Bayesian approach to the Lasso by placing a Laplace prior on β,

$$\pi_0(\beta \,|\, \sigma^2) = \left(\frac{\lambda}{2\sqrt{\sigma^2}}\right)^d e^{-\lambda|\beta|_1/\sqrt{\sigma^2}}.$$

We use their hierarchical model, which places on σ^2 the noninformative scale-invariant marginal prior, $\pi_0(\sigma^2) = 1/\sigma^2$. Thus, the full posterior for the Bayesian Lasso is

$$\begin{aligned}
\pi(\theta \,|\, X, \mathbf{y}) \propto \pi_0(\theta)\pi(X \,|\, \theta, \mathbf{y}) &= \pi_0(\beta, \sigma^2)\pi(X \,|\, \beta, \sigma^2, \mathbf{y}) \\
&= \pi_0(\sigma^2)\pi_0(\beta \,|\, \sigma^2)\pi(X \,|\, \beta, \sigma^2, \mathbf{y}) \\
&= \left(\frac{1}{\sigma^2}\right)\left(\frac{\lambda}{2\sqrt{\sigma^2}}\right)^d e^{-\lambda|\beta|_1/\sqrt{\sigma^2}} \left(\frac{1}{\sqrt{2\pi\sigma^2}}\right)^N e^{(\mathbf{y}-X\beta)^\top(\mathbf{y}-X\beta)},
\end{aligned}$$

where the likelihood term $\pi(X \,|\, \beta, \sigma^2, \mathbf{y})$ comes from the normal model $\mathbf{y} \sim \mathcal{N}(X\beta, \sigma^2)$. The parameter vector θ concatenates σ and β, thus it is 57-dimensional and real-valued. In our experiments, we calculate the log posterior as the sum of the log prior,

$$\begin{aligned}
\log \pi_0(\theta) = \log \pi_0(\beta, \sigma^2) &= \log \pi_0(\sigma^2) + \log \pi_0(\beta \,|\, \sigma^2) \\
&= \log\left(\frac{1}{\sigma^2}\right) + d\log\left(\frac{\lambda}{2\sqrt{\sigma^2}}\right) - \frac{\lambda|\beta|_1}{\sqrt{\sigma^2}},
\end{aligned}$$

setting $\lambda = 5.0$, and the log likelihood,

$$\log \pi(X, \mathbf{y} \,|\, \theta) = \log \pi(X, \mathbf{y} \,|\, \beta, \sigma^2) = N \log\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) + (\mathbf{y} - X\beta)^\top(\mathbf{y} - X\beta),$$

which is evaluated in batches as

$$\sum_{n=1}^{N} \log \pi(\mathbf{x}_n, y_n \,|\, \theta) = \sum_{n=1}^{N} \log \pi(\mathbf{x}_n, y_n \,|\, \beta, \sigma^2) = \sum_{n=1}^{N} \log\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) + (y_n - \mathbf{x}_n^\top \beta)^2.$$

5.2 Adaptive proposal distribution

In all of our experiments, we use a spherical, axis-aligned Gaussian for the proposal distribution, *i.e.*,

$$\theta' \sim q(\theta' \,|\, \theta) = \mathcal{N}(\theta' \,|\, \theta, \lambda^2 I_d), \tag{5.1}$$

where $\lambda \in \mathbb{R}^+$ is the standard deviation, I_d is the d-dimensional identity matrix and d is the dimension of θ. In our preliminary experiments, which we don't include in our evaluation here, we used a fixed proposal distribution. This was problematic because – as we discussed in Section 2.4.1 – the behavior of MH is both sensitive to the proposal distribution and changes over the course of execution. As a result, it was difficult to tune the parameters of the proposal distribution to yield experiments satisfying our requirements stated at the beginning of this chapter: each MH simulation starts away from convergence, progresses through burn-in and eventually converges, while achieving a meaningful acceptance rate and number of effective samples. Specifically, suppose we set the proposal distribution to achieve an acceptance rate of about 0.234 during the burn-in phase. MH advances until it is close to a mode of the target, but there the proposals tend to be far from the mode and thus have low probability. This results in a high rate of rejection and the algorithm becomes stuck. Alternately, if we tune the proposal distribution to sample well around such a mode of the target distribution, then the characteristic step size tends to be much smaller than before and progress is artificially slow during burn-in.

Our solution employs a simple adaptive scheme to set the parameters of the proposal distribution, improving convergence relative to standard MH. This approach falls under the provably convergent adaptive algorithms studied by Andrieu and Moulines (2006) and was easily incorporated into our framework. The general idea behind adaptive MH is to improve performance by tuning the proposal distribution during execution, using information from the samples as they are generated, in a way that converges asymptotically. Often, it is desirable for the proposal distribution to be close to the target. This motivates adaptive schemes that fit a distribution to the observed samples and use this fitted model as the proposal distribution. For example, a simple online procedure can update the mean μ and covariance Σ of a multidimensional Gaussian model as follows:

$$\begin{aligned}\mu_{t+1} &= \mu_t + \gamma_{t+1}(\theta_{t+1} - \mu_t) \qquad t \geq 0 \\ \Sigma_{t+1} &= \Sigma_k + \gamma_{t+1}((\theta_{t+1} - \mu_t)(\theta_{t+1} - \mu_t)^\top - \Sigma_t),\end{aligned}$$

where t indexes the MH iterations and γ_{t+1} controls the speed with which the adaptation vanishes. An appropriate choice is $\gamma_t = t^{-\alpha}$ for $\alpha \in [1/2, 1)$. The tutorial by Andrieu and Thoms (2008) provides a review of this and other, more sophisticated, adaptive MH algorithms.

Our adaptive scheme directly uses information about whether proposals are accepted or rejected to tune the proposal distribution to achieve an acceptance rate of approximately 0.234. Let ρ be a node in the MH binary tree. Denote by $\mathbf{1}_\rho$ the indicator for whether θ_ρ corresponds to an accepted or rejected state, *i.e.*,

$$\mathbf{1}_\rho = \begin{cases} 1 \text{ if } \rho \text{ is a right child in the MH binary tree} \\ 0 \text{ if } \rho \text{ is a left child in the MH binary tree.} \end{cases}$$

Our strategy is to increase λ, the scale of our proposal distribution in Equation 5.1, if the acceptance rate is too high and decrease it if the acceptance rate is too low. Our adaptive rule achieves this by modifying $\ell = \log \lambda^2$, the log of the variance, as follows:

$$\ell_{t+1} = \gamma_{t+1}(\mathbf{1}_\rho - 0.234).$$

We set $\ell_0 = \log(2.38^2/d)$, which corresponds to the proposal distribution with the "optimal" acceptance rate of 0.234, derived for the case where the target is a standard d-dimensional normal distribution, in the limits where the chain has converged and $d \to \infty$ (Roberts et al., 1997). We empirically found $\gamma_t = t^{-1/2}$ to work well. Our adaptive approach can be generalized to more complicated proposal distributions, but we did not need any for our experiments.

To support this adaptive MH algorithm within our prefetching framework, we made a simple extension to our system. In general, adaptive MH depends on the history of the simulated chain. Our adaptive scheme depends on the sequence of accepted and rejected states, *i.e.*, the chain's path through the MH binary state tree. Given an initial value for ℓ_0, the trajectory of ℓ_t is completely determined by this path. Thus, whenever we create a new node ρ in the jobtree, we generate the corresponding value of ℓ_ρ and store it on the node. This information, which is stored on the master, is communicated to a worker, via a `HAVE-WORK` message, when called upon to generate the proposal at ρ.

For our mixture of Gaussians problem, we follow the standard convention of additionally permuting the dimension labels each time a proposal is generated. In the Bayesian Lasso problem, the first coordinate of θ is a standard deviation and must be positive, so we truncate this dimension of the proposal distribution accordingly.

5.3 Assessing chain convergence and quality

We assess chain convergence using the Gelman-Rubin statistic known as $\hat{R}$ (Gelman and Rubin, 1992); the description here follows that in their classic textbook (Gelman et al., 2003). Suppose we run S separate chains such that each produces T samples. Let $\theta_{ts} \in \mathbb{R}^d$ refer to sample t in chain s. Let $\psi_{ts} = f(\theta_{ts})$ where $f : \mathbb{R}^d \to \mathbb{R}$ is some scalar function of θ_{ts}, *e.g.*, f could be the log posterior, or alternatively, the first coordinate of θ_{ts}. First, we compute the between-chain variance

$$B = \frac{T}{S-1}\sum_{s=1}^{S}(\bar{\psi}_{\cdot s} - \bar{\psi}_{\cdot\cdot})^2, \quad \text{where} \quad \bar{\psi}_{\cdot s} = \frac{1}{T}\sum_{t=1}^{T}\psi_{ts} \quad \text{and} \quad \bar{\psi}_{\cdot\cdot} = \frac{1}{S}\sum_{s=1}^{S}\bar{\psi}_{\cdot s},$$

and the within-chain variance

$$W = \frac{1}{S}\sum_{s=1}^{S}\delta_s^2, \quad \text{where} \quad \delta_s^2 = \frac{1}{T-1}\sum_{t=1}^{T}(\psi_{ts} - \bar{\psi}_{\cdot s})^2.$$

Now we can estimate the marginal posterior variance of ψ as

$$\nu = \frac{T-1}{T}W + \frac{1}{T}B.$$

The estimate of the scale of the distribution of ψ is then $\sqrt{\nu}$. Notice that $\lim_{T\to\infty}\nu = W$. This makes sense because each chain asymptotically samples from the correct distribution. Furthermore, $\nu > W$ whenever $B > W$, which tends to be true before the chains have converged, *i.e.*, differences between samples from different chains are greater than differences within chains. The quantity $\hat{R} = \sqrt{\nu/W}$ estimates the amount by which this scale would decrease if the simulations were continued to the limit $T \to \infty$. Notice that in this limit, $\hat{R}$ converges to 1. Furthermore, $\hat{R}$ tends to decrease toward 1, following the above reasoning about ν. A common heuristic is to consider values of $\hat{R} < 1.1$ as acceptable; lower cut-off values are considered better.

In our experiments, $S \geq 2$ and we run each chain for at least 50000 iterations. We define $\psi_{ts}^{(i)}$ to be the ith coordinate of $\theta_{ts} \in \mathbb{R}^d$ and assess convergence for each dimension of θ separately; define $\hat{R}^{(i)}$ to be $\hat{R}$ evaluated for the ith coordinate. Our objective is to identify a point at which $\hat{R}^{(i)}$ reaches a reasonable value across all dimensions of θ. For each dimension, we compute $\hat{R}^{(i)}$ for increasingly longer subsequences of $\psi_{\cdot s}^{(i)}$. We consider subsequences of length L starting at $t = 1$ and always discard the first half, thus $T = L/2$; this sort of discarding of samples is another commonly used guideline. Using the second half of the subsequence, we compute $\hat{R}^{(i)}$ for increasing values of L, until we observe $\hat{R}^{(i)} < 1.05$ for all dimensions $i = 1, \dots, d$. We define *burn-in* to be the period before we observe this to be true.

We also assess the quality of the samples obtained after burn-in. For each dimension, we measure the effective number of samples, defined by Gelman et al. (2003) as $n_{\text{eff}} = ST\nu/B$. We consider increasingly shorter subsequences of samples that start at varying points after burn-in and extend to the end of the experiment; we do not discard any additional samples. We note that n_{eff} does not monotonically decrease as we consider these shorter subsequences. Therefore, we identify a subsequence of samples that approximately maximizes the average value of n_{eff} across dimensions.

5.4 Speedup results

We evaluate our system with up to 64 worker cores in a multicore cluster environment in which machines are connected by 10Gb ethernet and each machine has 32 cores (four 8-core Intel Xeon E7-8837 processors).

We expect predictive prefetching to perform best when the densities at a proposal and corresponding current point are significantly different, which is common in the initial burn-in phase of chain evaluation. In this phase, early estimates based on small subsamples effectively predict whether the proposal is accepted or rejected. When the density at the proposal is close to that at the current point – for example, as the proposal distribution approaches the target distribution – the outcome is inherently difficult to predict; early estimates will be uncertain or even wrong. Incorrect estimates could destroy speedup (no precomputations would be useful). We hope to do better than this worst case, and to at least achieve logarithmic speedup. In our experiments, we divide the evaluation of the target function into 100 batches. Thus, for the Gaussian mixture problem, each subsample contains 10^4 data items, and for the Bayesian Lasso problem, each subsample contains 1.8×10^4 data items.

	Burn-in					
J	$i_1 = 9575$		$i_2 = 24000$		$i_3 = 50000$	
1	16674	—	41978	—	87500	—
16	2730	6.1×	8678	4.3×	20318	4.3×
32	1731	9.6×	7539	5.6×	19046	4.6×
64	989	16.8×	5894	7.1×	15146	5.8×

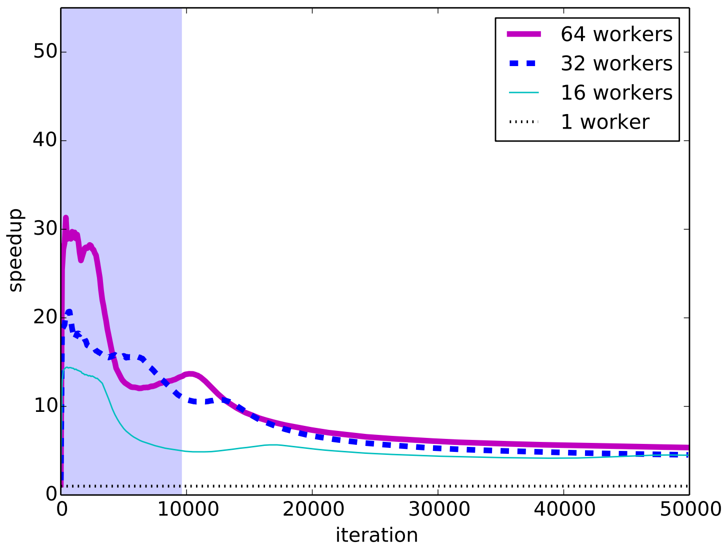
Table 5.1: Cumulative time (in seconds) and speedup for evaluating the Gaussian mixture model with different numbers of workers J.

	mean	standard deviation	min	max
n_{eff}	3405	7253	50	26000
$\hat{R}$	1.005	0.006	1.000	1.020

Table 5.2: Convergence statistics after burn-in (over iterations i_2–i_3) for the Gaussian mixture model, computed over the 64 dimensions of the model.

Table 5.1 shows the results for the Gaussian mixture model. We run the model with the same initial conditions and pseudorandom sequences with varying numbers of worker threads. All experiments produce identical chains. We evaluate the cumulative time and speedup obtained at three different iteration counts. The first, $i_1 = 9575$ iterations, is burn-in. After i_1 iterations, all dimensions of samples achieve the Gelman-Rubin statistic $\hat{R} < 1.05$, computed using two independent chains, where the first $i_1/2$ samples have been discarded (Gelman and Rubin, 1992). We then run the model further to i_3 iterations. Iterations $i_2 = 24000$ through $i_3 = 50000$ are used to compute an effective number of samples n_{eff}. (Table 5.2 shows convergence statistics after i_3 iterations.) The results are as we hoped. The initial burn-in phase obtains better-than-logarithmic speedup (though not perfect linear speedup). With 64 workers, the chain achieves burn-in 16.8× faster than with one worker. After burn-in, efficiency drops as expected, but we still achieve logarithmic speedup (rather than sub-logarithmic). At 50000 iterations, speedup for each number of workers J rounds to $\log_2 J$.

Figure 5.1 explains these results by graphing cumulative speedup over the whole range of iterations. The initial speedup is good – we briefly achieve more than 30× or 40× speedup, depending on the initial condition, at $J = 64$ workers. As burn-in proceeds, cumulative speedup



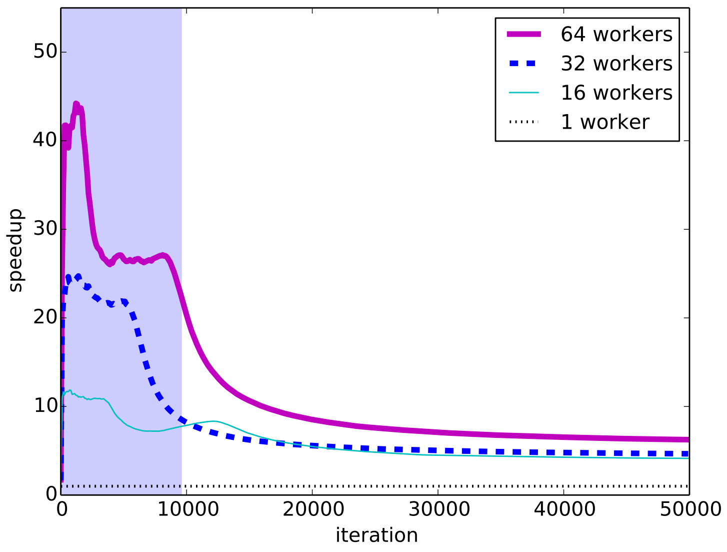


Figure 5.1: Cumulative speedup relative to our baseline, as a function of the number of MH iterations, for the mixture of Gaussians problem. The two figures correspond to different initial conditions, and the different curves correspond to different numbers of workers. Pale blue shading highlights the burn-in phase, *i.e.*, the first $i_1 = 9575$ iterations.

falls off to logarithmic in J. Figure 5.2 shows cumulative speedup for the Gaussian mixture model with several different initial conditions. Each initial condition is drawn from the same generative model as the model parameters, as described in Section 5.1.1. We see a range of variation due to differences in the adaptive scheme during burn-in. The overall pattern is stable, however: good speedup during burn-in followed by logarithmic speedup later. Also note

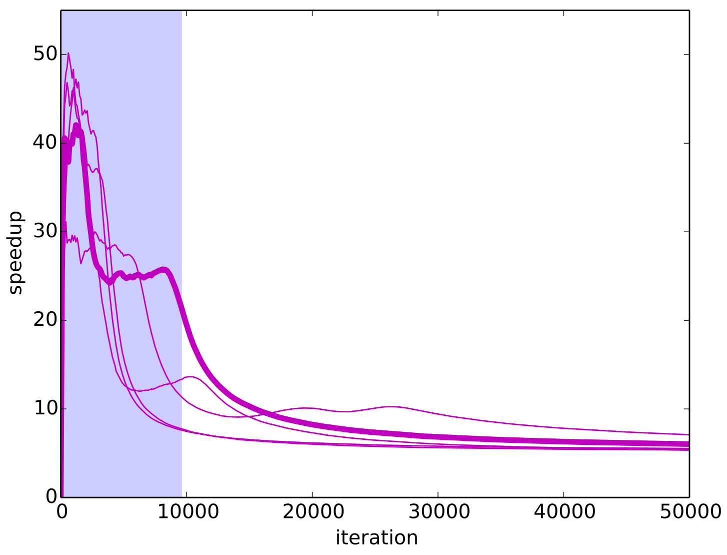
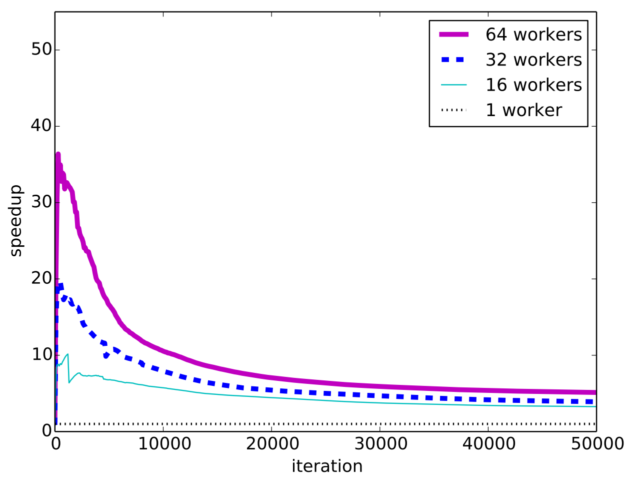
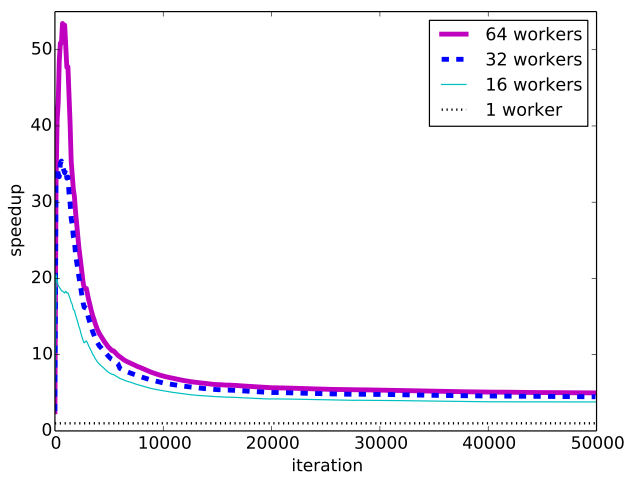


Figure 5.2: Cumulative speedup relative to our baseline, as a function of the number of MH iterations, for the mixture of Gaussians problem. The different curves correspond to different initial conditions; all curves are for 64 workers. Pale blue shading highlights the burn-in phase, *i.e.*, the first $i_1 = 9575$ iterations.

that speedup does not necessarily decrease steadily, or even monotonically. At some initial conditions, the chain enters an easier-to-predict region before truly burning in; while in such a region, speedup is maintained. Our system takes advantage of these regions effectively.

Figure 5.3 shows that good speedups are achievable for real problems. The speedup behavior for the Bayesian Lasso problem appears similar to that of the mixture of Gaussians. There are differences, however: Lasso evaluation did not converge by 50000 iterations according to standard convergence statistics. On several initial conditions, the chain started taking small steps, and therefore dropped to logarithmic speedup, before achieving convergence. Overall performance might be improved by detecting this case and switching some speculative resources over to other initial conditions, an idea we leave for future work.





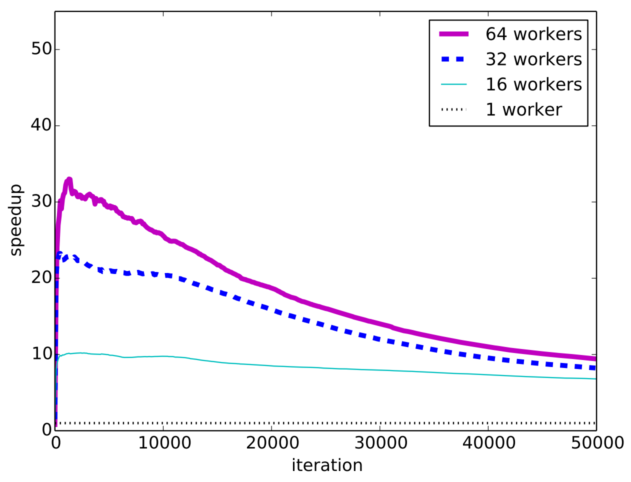
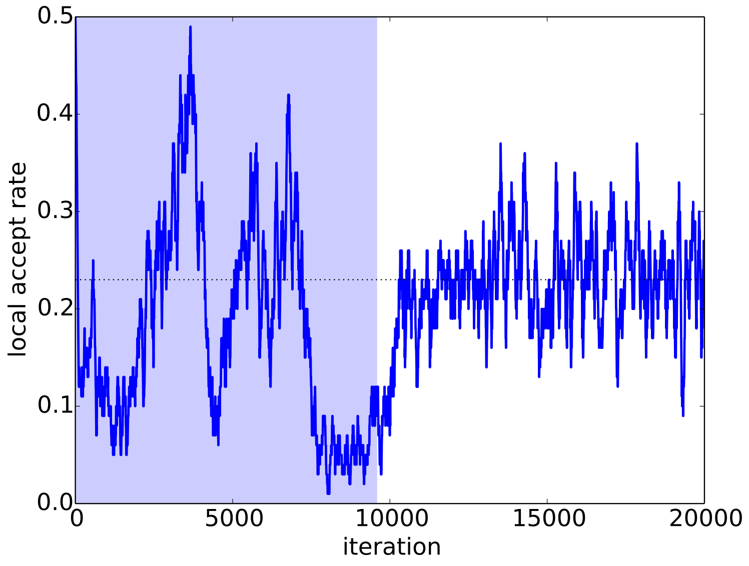


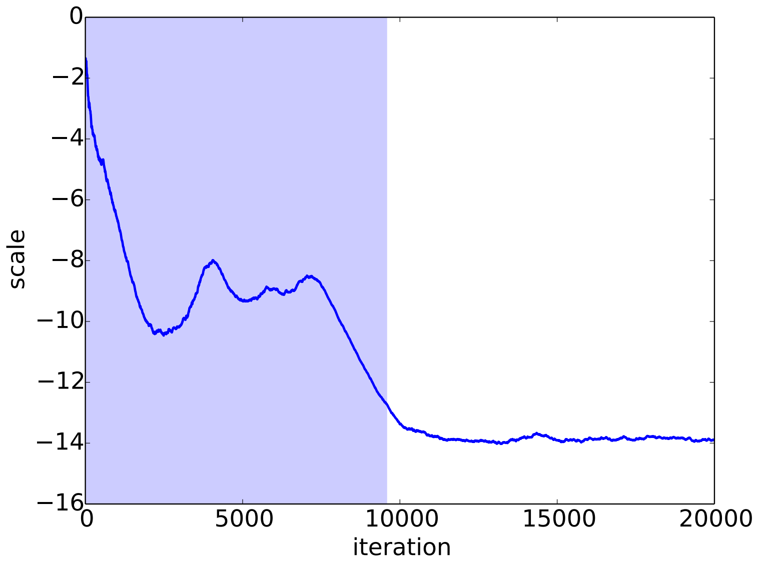
Figure 5.3: Cumulative speedup relative to our baseline, as a function of the number of MH iterations, for the Bayesian Lasso problem. The different curves correspond to different numbers of workers. The different figures are for different initial conditions.

5.5 Adaptive Metropolis–Hastings behavior

Figure 5.4 illustrates the behavior of our adaptive Metropolis–Hastings algorithm for the mixture of Gaussians problem. This procedure, described in Section 5.2, adaptively tunes the proposal distribution to achieve an acceptance rate of 0.234. Specifically, it tunes $\ell = \log \lambda^2$, where λ is the scale of the spherical Gaussian proposal distribution. Note that the adaptation is not affected by prefetching. Figure 5.4a plots a trace of the local acceptance rate, which we defined in Section 4.4 to be the empirical acceptance rate observed during the simulation of the k most recent MH samples. In our experiments, $k = \min\{t, 100\}$, where t is total number of MH samples obtained thus far. During burn-in, the local acceptance rate varies broadly, nearly over the entire range of $[0.0, 0.5]$, and afterward settles around the target value of 0.234. Recall that we define burn-in as the first $i_1 = 9575$ iterations, as described in Section 5.4 and reported in Table 5.1. Figure 5.4b plots the trajectory of the adapted parameter. As expected, the values of ℓ are larger during burn-in – when proposals can be made father away without suffering from rejection – than afterward. From its initial value, ℓ generally decreases during burn-in, though not monotonically, until it stabilizes to a small value after convergence.



(a) Adaptation during execution of the local acceptance rate.



(b) Adaptation during execution of the proposal distribution's scale parameter, $\ell = \log \lambda^2$.

Figure 5.4: Behavior of our adaptive Metropolis–Hastings algorithm, which (a) achieves the target acceptance rate of 0.234 by (b) tuning the proposal distribution. Pale blue shading highlights the burn-in phase, after which the local acceptance rate settles around the target value and the proposal scale parameter stabilizes.

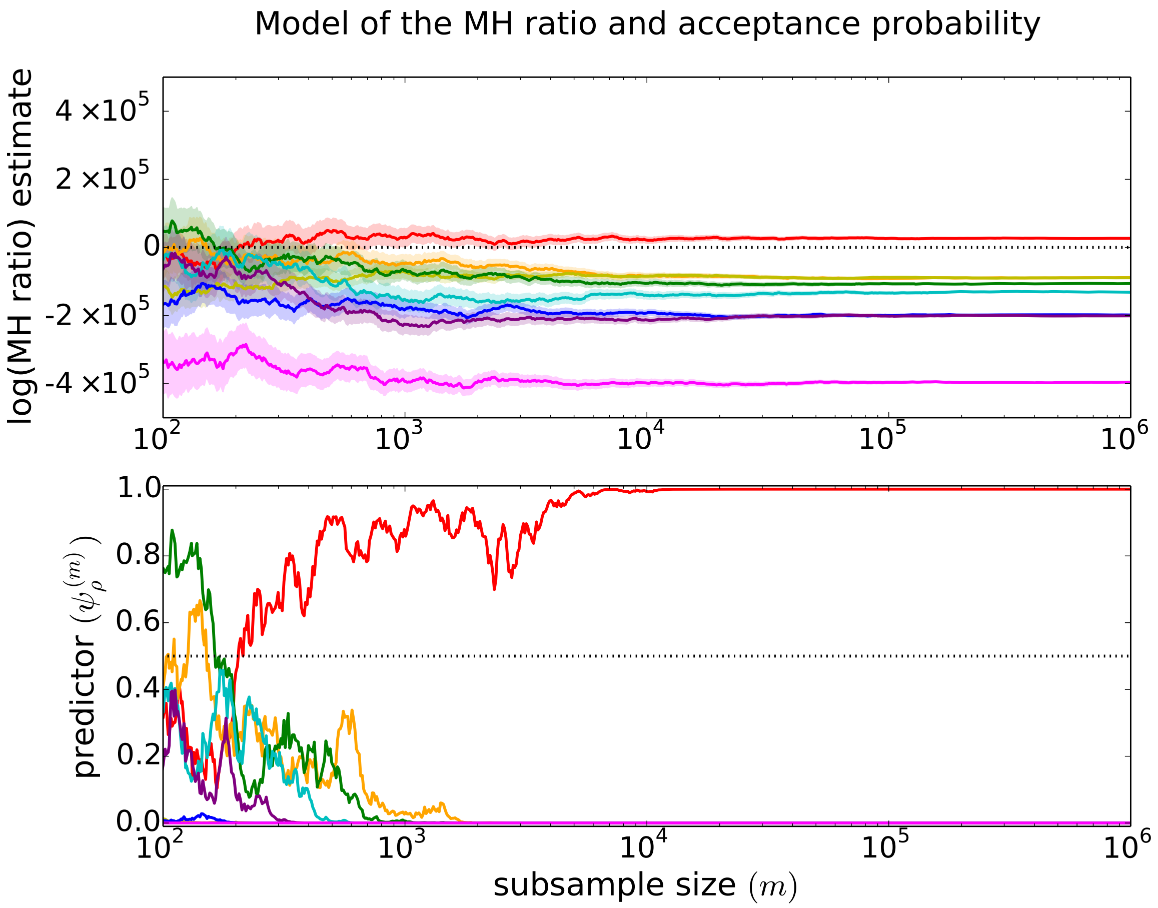


Figure 5.5: Example predictor trajectories for the mixture of Gaussians problem during burn-in. The upper subfigure plots the estimate of log of the MH ratio as a function of subsample size m. The shaded region around each trace indicates one standard deviation in our error model. The lower subfigure plots the predictor $\psi_\rho^{(m)}$ as a function of m. Different colors indicate different (θ, θ') pairs. Each set of traces corresponds to a sequence of MH iterations.

5.6 Estimate, error model and predictor behavior

In this section, we describe the behavior of our predictor for the mixture of Gaussians problem. The predictor depends on the estimate for the log of the Metropolis–Hastings ratio, the normal error model for this estimate and a uniform random variate u. Figures 5.5 and 5.6 show the behavior of the estimate, its error and the subsequent predictors (for randomly chosen u) during and after burn-in, respectively. At the beginning of burn-in, estimates are

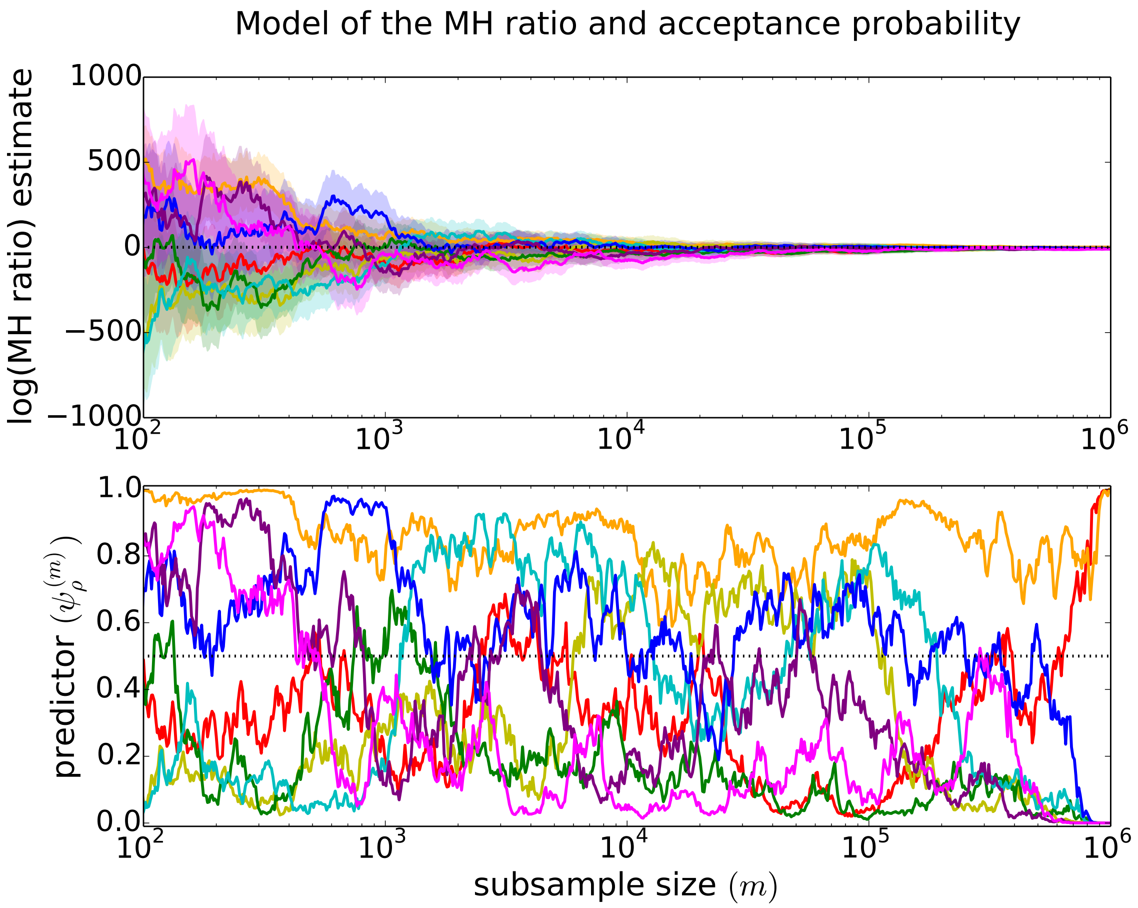


Figure 5.6: Example predictor trajectories for the mixture of Gaussians problem after burn-in. The upper subfigure plots the estimate of log of the MH ratio as a function of subsample size m. The shaded region around each trace indicates one standard deviation in our error model. The lower subfigure plots the predictor $\psi_\rho^{(m)}$ as a function of m. Different colors indicate different (θ, θ') pairs. Each set of traces corresponds to a sequence of MH iterations.

effective, and the predictor converges quite quickly to the correct (final) indicator. After burn-in, the new proposal's target density is close to the old proposal's, and the estimates are similarly hard to distinguish. Notice that the scale of the log of their ratio is orders of magnitude smaller after burn-in compared to the beginning of burn-in. The random variate u could be small enough for the predictor to converge quickly to 1; more often, the predictor varies widely over time, and does not converge to 0 or 1 until almost all data are evaluated. This behavior makes logarithmic speedup a best case. Luckily, the predictor is more typically uncertain (with an intermediate value) than wrong (with an extreme value that eventually

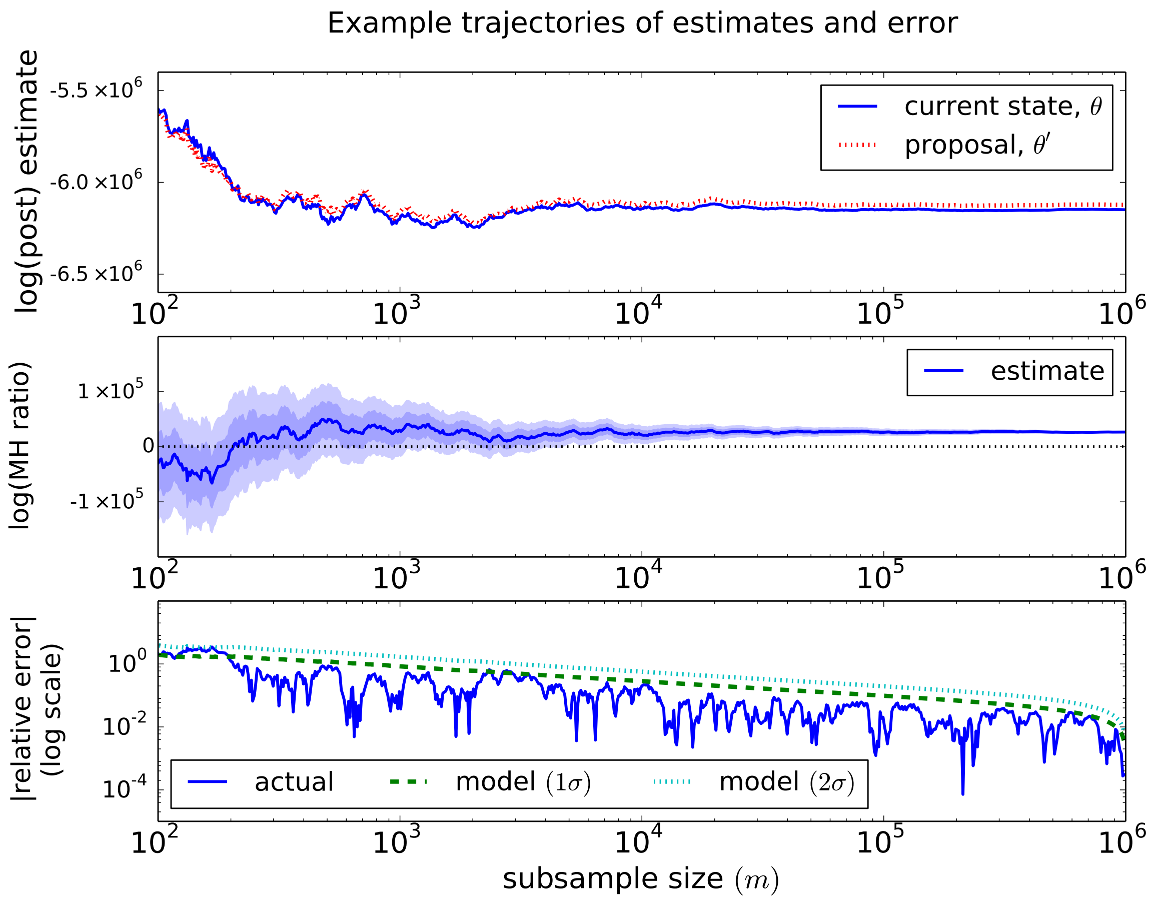


Figure 5.7: Estimates and error during burn-in. Each subfigure plots traces as a function of subsample size. The upper subfigure plots the estimates of the log posterior at the current state (solid blue) and proposal (dotted red). The middle subfigure plots the estimate of the log MH ratio (solid line), with shaded regions indicating one (dark) and two (light) times the standard error. The lower subfigure plots on a log scale the absolute error of this estimate relative to the true (final) value (solid blue), as well as one (dashed green) and two (dotted cyan) times the standard error.

flips to the opposite value): incorrect predictors could lead to sublogarithmic speedup.

Our estimates depend on the order in which the data are evaluated. In general, we might be worried about malicious orderings of the data, which could lead to biased estimates, bad predictors and performance degradation. In our experiments, we permute the data once at the very beginning. A more sophisticated solution would be to occasionally re-permute the data during execution, *e.g.*, every 20 iterations or so. Our current implementation could be

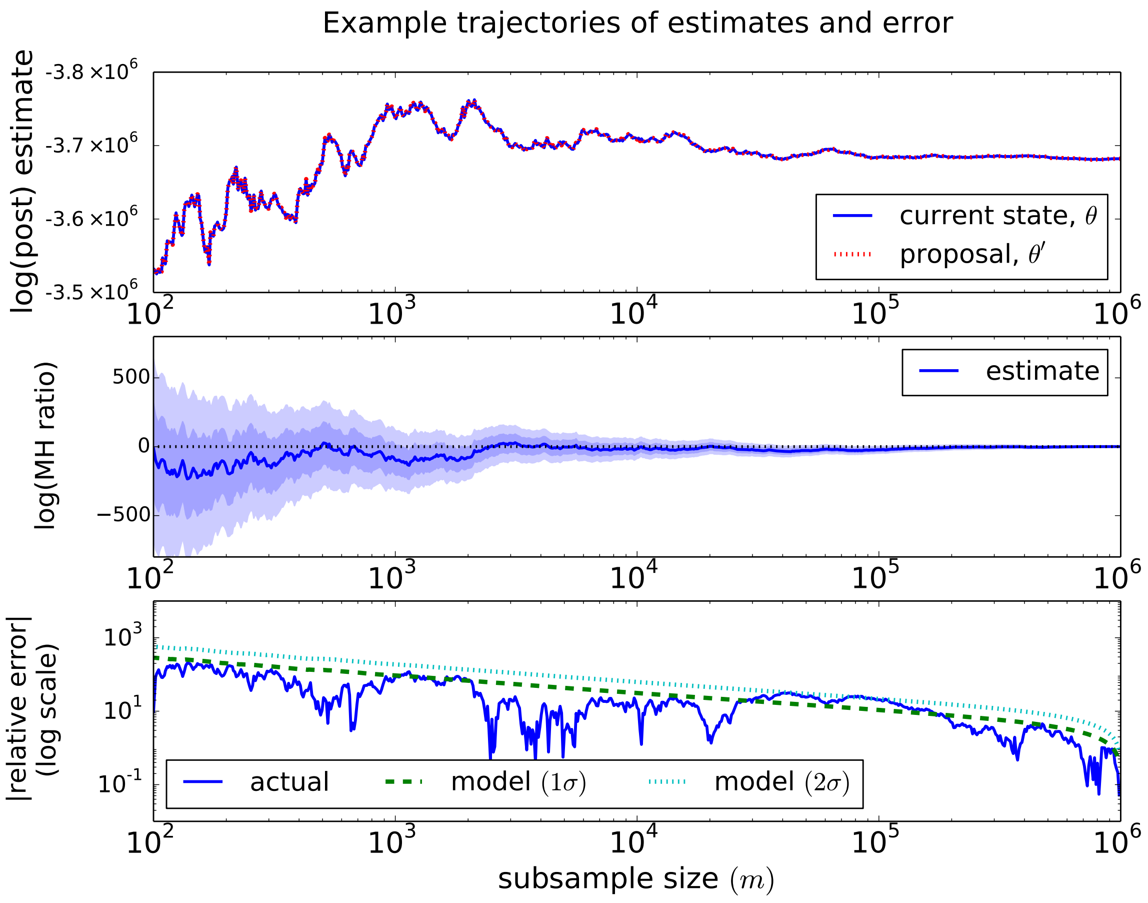


Figure 5.8: Estimates and error at convergence, analogous to Figure 5.7.

modified to support this, but care would be required to avoid hurting performance. For example, suppose we decide to permute the data after accepting state θ'. At the next iteration, MH compares θ to θ', but each is now associated with a different ordering of the data. This is unfavorable because our predictors work best when the data are evaluated in the same order for both θ and θ'.

Figures 5.7 and 5.8 illustrate, in greater detail, the evolution of the MH ratio estimate, during and after burn-in, respectively. Each upper subfigure separately plots the estimates of the log posterior at the current state and the proposal. These traces are highly correlated, since the log likelihoods at the current state and proposal are highly correlated for each datum. At the beginning of burn-in, the correlation between $\log \pi(\theta \,|\, x_i)$ and $\log \pi(\theta' \,|\, x_i)$

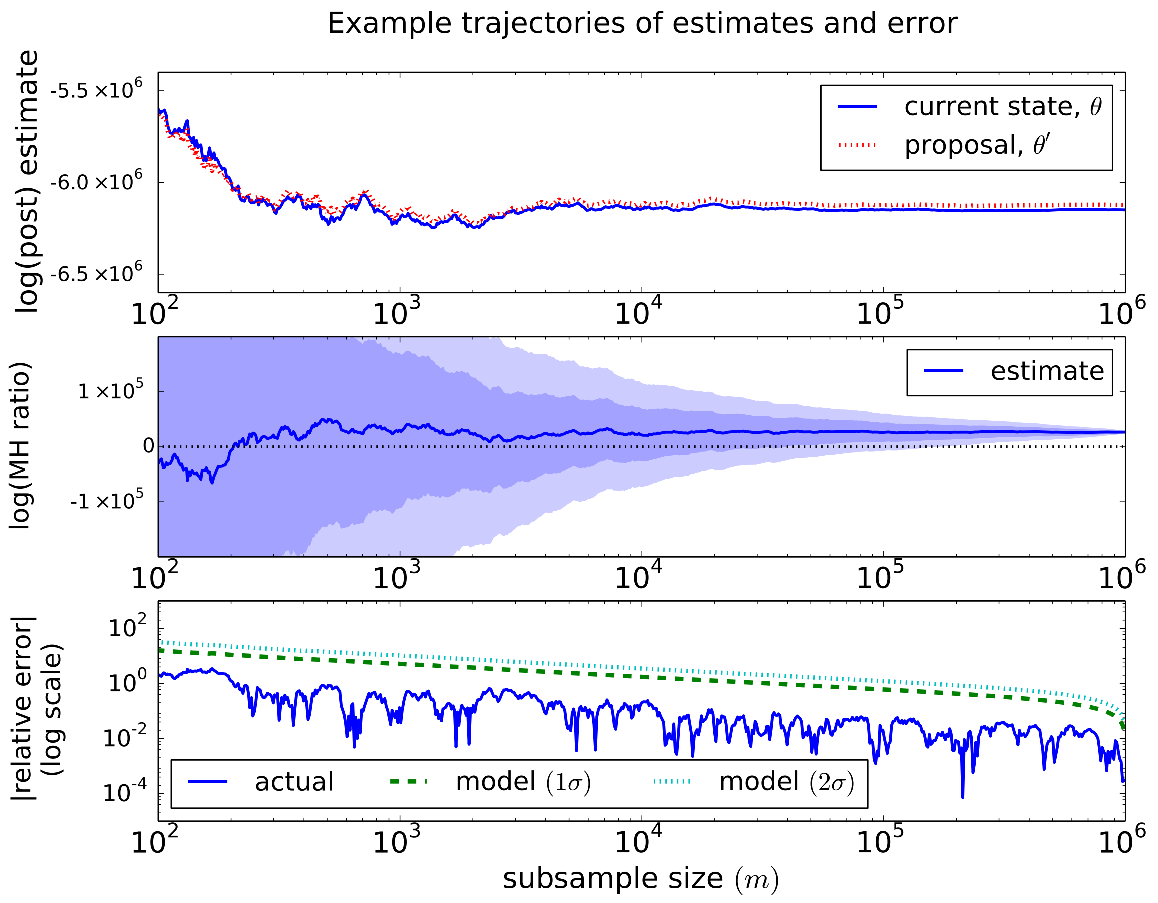


Figure 5.9: Estimates and naïve error model during burn-in. The estimates are the same as in Figure 5.7, but the error is significantly worse. Note that the scales of all the axes are the same as those in Figure 5.7.

is greater than 0.9; after convergence, it is greater than 0.9999. This summarizes why it becomes more difficult to predict whether a proposal will be accepted or rejected – the target evaluations at θ and θ' are practically indistinguishable. Each middle subfigure plots the corresponding estimate of the log MH ratio, which cross the dotted line at zero whenever the above estimates cross. Shading around the estimate corresponds to our error model. Each lower subfigure plots the error of this estimate relative to the true (final) value on a log scale; our model is consistent with the actual error.

Our choice of error model has a significant impact on the predictors we form to make scheduling decisions. Figure 5.9 illustrates a naïve approach, briefly mentioned in Section 3.4,

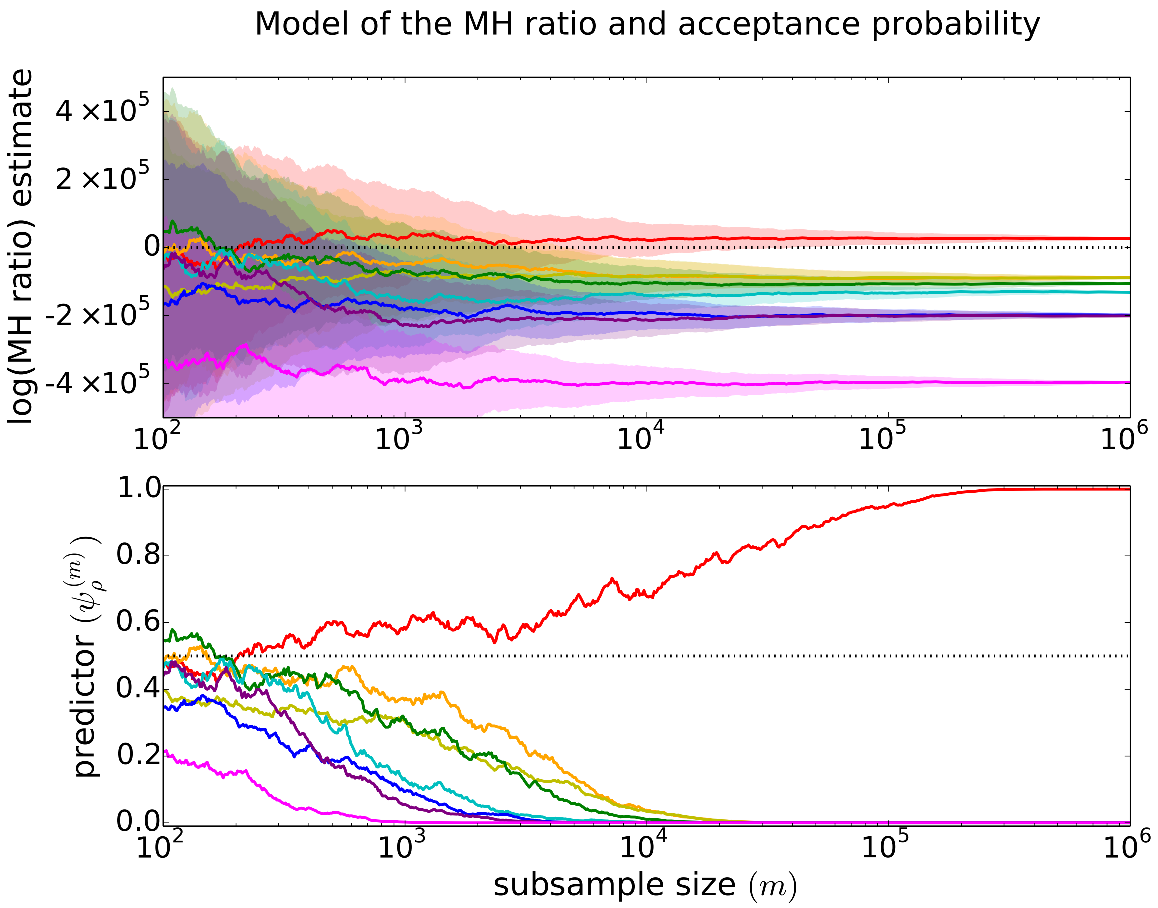
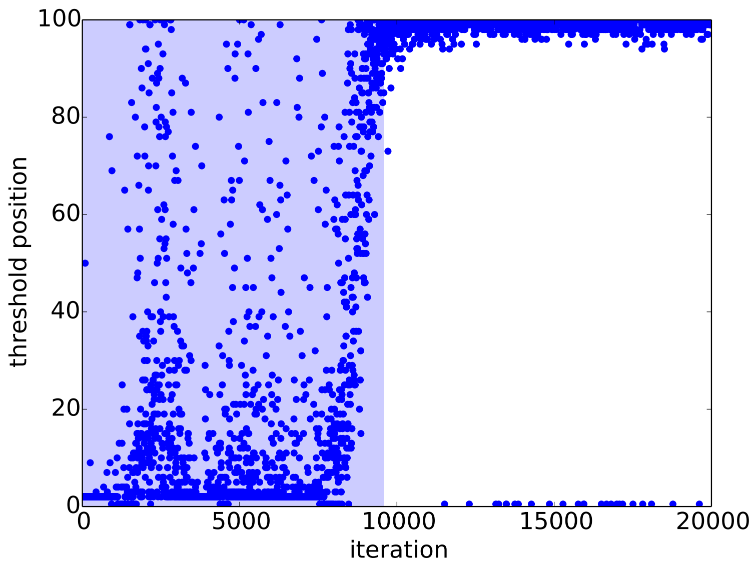


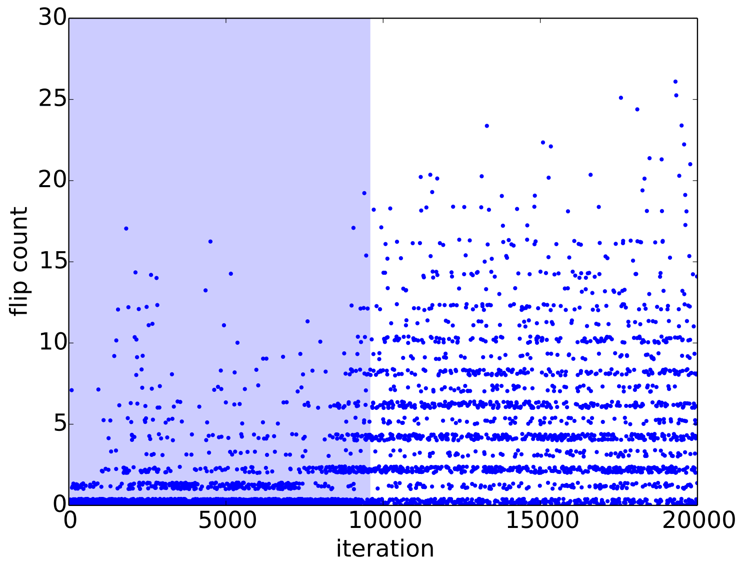
Figure 5.10: Example predictor trajectories for the mixture of Gaussians problem during burn-in. The trajectories and estimates are the same as in Figure 5.5, but here we use the naïve error model, as in Figure 5.9.

that separately models the estimates for the log posteriors at θ and θ' without capturing the correlation between $\log \pi(\theta \mid x_i)$ and $\log \pi(\theta' \mid x_i)$. As expected, the estimates for the full log posteriors are the same as before, but the estimated error is dramatically larger. This translates to inaccurately high uncertainty in the estimates and would result in needlessly conservative speculation. This is illustrated in Figure 5.10, which by comparison to Figure 5.5 shows that the naïve error model results in predictors that must evaluate about an order of magnitude more data to converge. Note that these examples are representative of the beginning of burn-in; we expect the predictors to require even more data to converge as the execution progresses. Thus with the naïve error model, we would expect predictive prefetching to yield smaller benefits.

To see how predictor behavior changes over the course of execution, we track two quantities during execution that summarize the trajectory of each predictor concerned with an accept/reject decision on the true Markov chain path. Figure 5.11a plots the *threshold position*, which we define as the smallest number of batches evaluated after which the predictor does not cross 0.5, *i.e.*, one greater than the last batch leading to an incorrect prediction. Recall that in our experiments, the total number of batches is 100. The threshold position tends to increase over the course of execution, with a relatively sharp phase transition from burn-in to convergence. After convergence, essentially all the data must be inspected. Figure 5.11b plots the *flip count*, which we define as the number of times the predictor crosses 0.5, *i.e.*, the number of times it "changes its mind" about whether the proposal will be accepted or rejected. The flip count also tends to increase over the course of execution, though it doesn't exhibit quite the dramatic transition as the threshold position. Both threshold position and flip count exhibit behavior consistent with Figures 5.5 and 5.6.



(a) Threshold position.



(b) Flip count.

Figure 5.11: Summaries of predictor behavior. Note that the data have been downsampled by a factor of five, and pale blue shading highlights burn-in. (a) The threshold position tends to increase over the course of execution. (b) The flip count tends to increase over the course of execution. For visual clarity, we have added a small amount of jitter, distributed uniformly and randomly on $[0, 0.4)$, to the integer-valued counts.

5.7 System measurements

In the previous section, we presented our main evaluation in terms of speedup. Here, we explain these results through measurements that characterize our system's behavior. As expected, we do not achieve perfect speedup, and the dominant reason is that only some of the speculative computation performed by workers is *useful*; any extra computation is not useful and we call it *wasted*. Figure 5.12 illustrates the distribution of useful (blue) and wasted (light gray) work on the Metropolis–Hastings binary tree, for a particular simulation path corresponding to the true output (thick arrows). Ignoring communication and other system overheads, perfect speedup is achieved when computation is performed only at the useful nodes. Lower *efficiency* is the result of wasted computation at other nodes and is given by the fraction of total computational time spent on useful work. Ideally, efficiency would equal 1, leading to perfect speedup. In practice, efficiency is less than 1 due to wasted work and other overheads. The measurements below explain these inefficiencies. In this section we focus on the mixture of Gaussians problem with 64 workers, specifically, the experiment using the second initial condition shown in Figure 5.1, with $N = 10^6$ data in the likelihood, evaluated in 100 batches.

During execution, our system exhibits different phases of behavior. Figure 5.13 plots overall progress as a function of wall clock time. There are two main phases: during and after burn-in. Progress is roughly constant within each phase and at least three times faster during burn-in than afterward; at least the first 10% of burn-in is even faster. From this trace, we choose iteration counts representative of these three phases: 500, 5000 and 15000. Below, we present a decomposition of how computational resources are utilized, at each of these iteration counts, on both the workers and the master.

Over the course of execution, we measured the total time all worker cores spent on five different tasks: generating useful proposals, wasted proposals, useful target evaluation, wasted target evaluation and waiting for a work assignment from the master. Figure 5.14 summarizes these results by plotting the fraction of time spent performing useful work (blue),

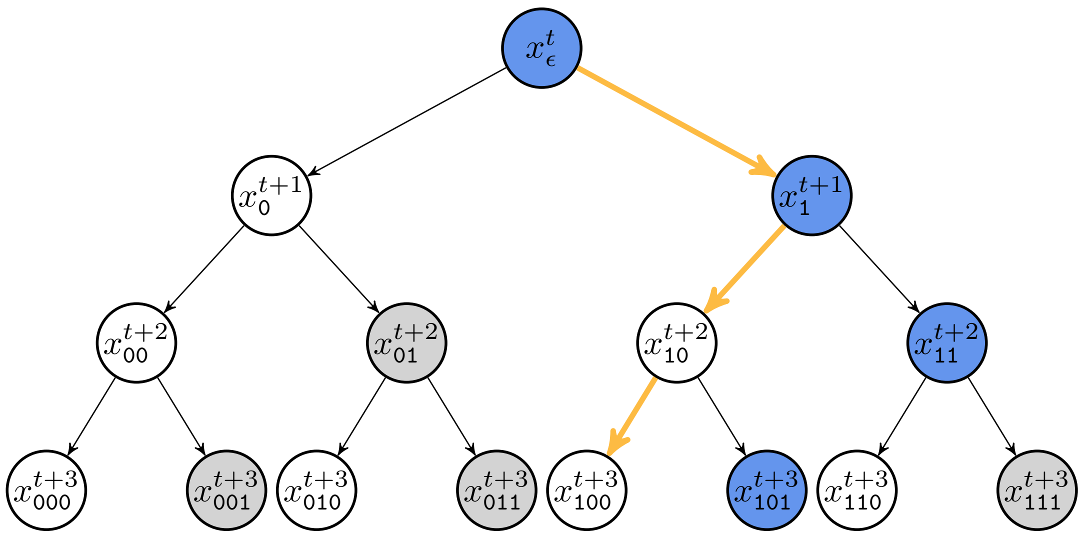


Figure 5.12: Metropolis–Hastings binary tree. Suppose that the thick arrows connect samples output by the algorithm. The blue circles then highlight nodes where computation must be performed, corresponding to useful work. When a prefetching scheme performs computation at the light gray nodes, this wasted work does not advance computation. Each remaining node is a copy of its parent and thus does not demand new computation.

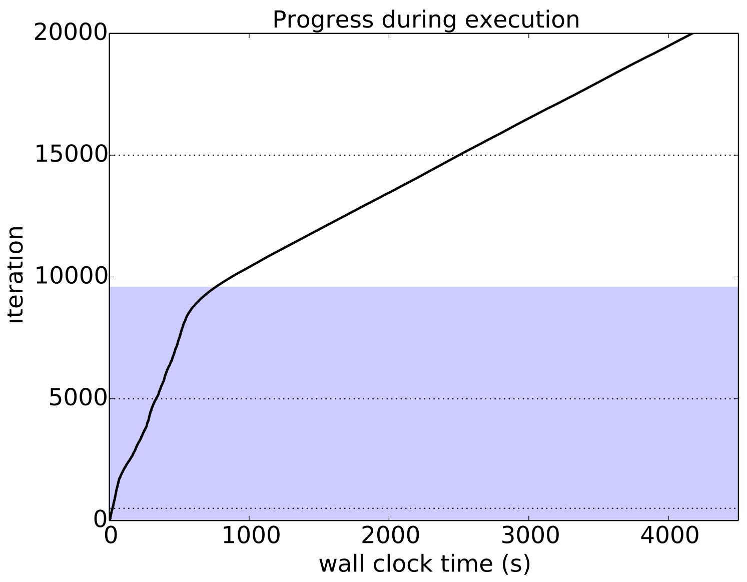


Figure 5.13: Progress as a function of wall clock time, in seconds, for the mixture of Gaussians problem with 64 workers. This experiment is the same as the second in Figure 5.1. The dotted horizontal lines are at 500, 5000 and 15000 iterations and correspond to the three iteration counts highlighted in Figures 5.14 and 5.15. We plot the progress out to 20000 iterations; the behavior remains stable for the remainder of the experiment, out to 50000 iterations.

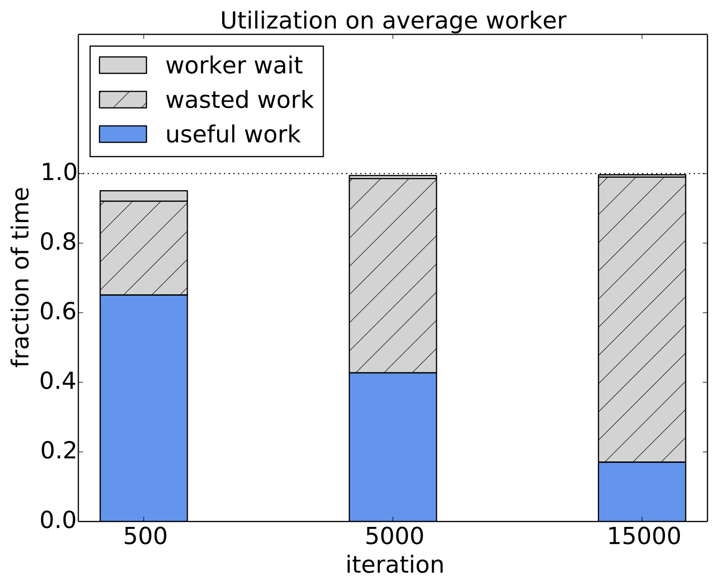


Figure 5.14: Cumulative fraction of time an average worker (64 total) spent performing useful (blue) or wasted (light gray, hatched) work, and waiting for work (light gray), normalized with respect to wall clock time, at three representative iteration counts. These measurements do not include the initial start-up time before each worker sends its first `WANT-WORK` message. After 500 iterations, this corresponds to less than 5% of the elapsed time.

wasted work (light gray, hatched) and waiting (light gray, solid). The fraction of time spent on useful work corresponds to efficiency and decreases as execution progresses. Figure 5.15 shows utilization on the master, divided into time spent acting in response to worker messages (blue) and time spent waiting for worker messages (light gray). Utilization on the master is stable for the entire execution; the master is active for less than 5% of the time.

Finally, Figure 5.16 plots the number of *allocated* jobtree nodes, *i.e.*, nodes explicitly represented in the jobtree stored on the master, over the course of execution, for 64 workers. We record this number at the end of each MH iteration. During burn-in, there are usually hundreds of allocated jobtree nodes, spanning less than 100 to greater than 500 during this time. In this phase, the prefetching is more aggressive, leading to irregular but deeper tree shapes that grow in the number of allocated nodes as predictions change. The number of allocated nodes decreases, sometimes sharply, whenever portions of the jobtree are trimmed. After convergence, the number of allocated nodes stabilizes to a much smaller range around 64,

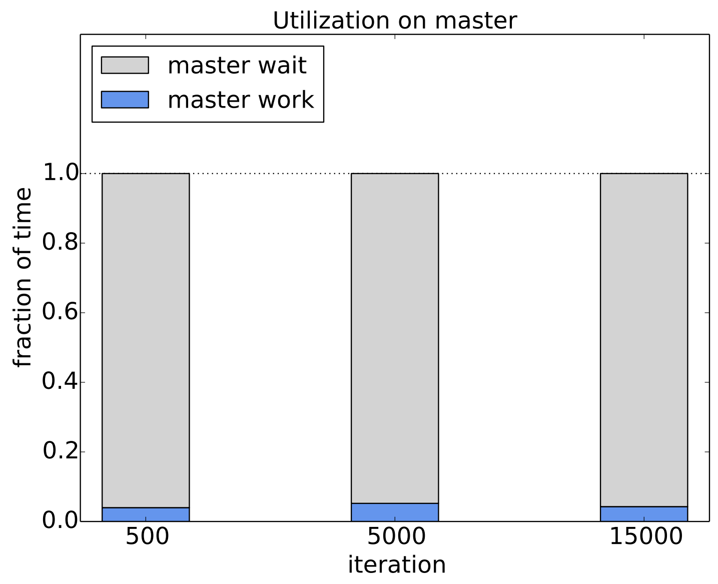


Figure 5.15: Cumulative fraction of time on master spent acting in response to messages from 64 workers (blue) and waiting for worker messages (light gray), normalized with respect to wall clock time, at three representative iteration counts.

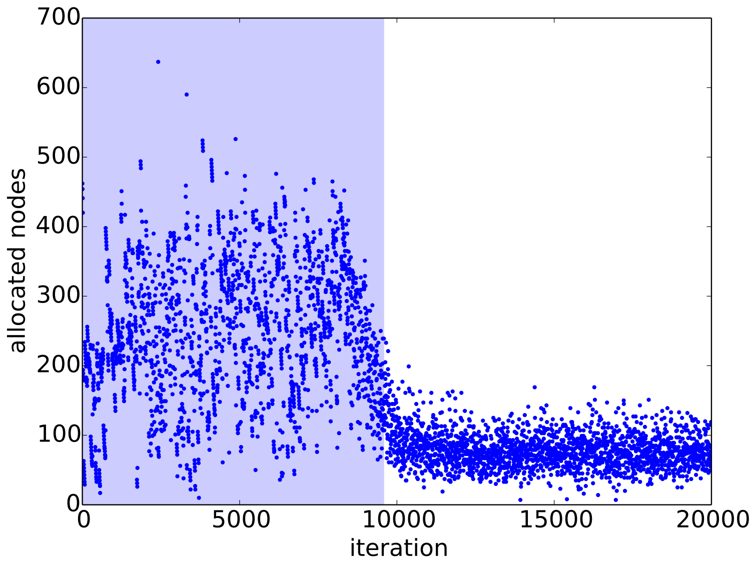


Figure 5.16: The number of allocated jobtree nodes during execution. Note that the data have been collected upon completion of each iteration and downsampled by a factor of five.

the number of workers. Then, the predictions are more or less ambiguous, and our prefetching scheme eventually looks more like the naïve scheme that densely allocates nodes starting at the root of the tree.

5.8 System overheads

The primary bottleneck in our implementation is due to the fan-in at the master. Specifically, our performance is sensitive to the rate at which the master must process messages from the workers, which scales with both the number of workers and the number of batches per target function evaluation. When the master is overwhelmed by messages, the workers end up waiting for work assignments, which decreases the fraction of time they spend on useful work. In Section 5.7, we summarized utilization on an average worker and the master, for the mixture of Gaussians problem with 64 workers, where the likelihood is evaluated in 100 batches. In this case, the average worker waits for less than 1% of the time and the master works for less than 5% of the time. Figures 5.17 and 5.18 illustrate the behavior for the same problem,[2] where we have decreased the batch size by a factor of 10, *i.e.*, increased to 1000 batches per likelihood evaluation. By 10000 iterations, the average worker spends about 10 times longer waiting (8% of the time) and the master spends 5 to 6 times longer working (25% of the time).

We could address this issue in several ways: decreasing the number of batches per target evaluation, eliminating the need for `WANT-WORK` messages and dividing the work of the master among multiple cores. In our current design, we use a constant batch size for our updates. However, this does not reflect information from our error model, which characterizes predictor uncertainty. For example, once we are relatively confident that a predictor has converged, then there isn't much advantage to sending updates in batches. Alternatively, when the error model indicates high uncertainty, the predictions carry little weight and not much information is gained until essentially all the data are evaluated. With larger batch sizes, we would probably want workers to periodically check for `ABANDON` messages during the evaluation of the batch – recall that these can be triggered by any update in the jobtree associated with any ancestral node. Also currently, the master does not assign work to a worker until it receives a `WANT-WORK` message, even though the master knows via `UPDATE` messages

[2]We note that the initial conditions are different.

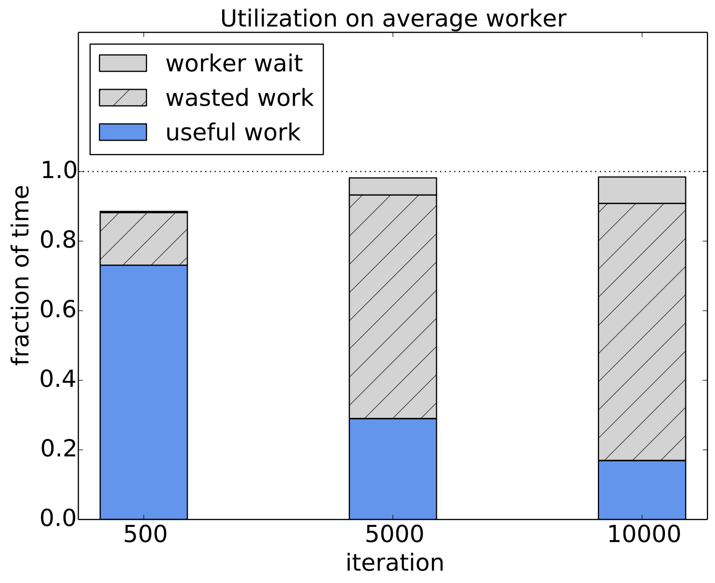


Figure 5.17: Cumulative fraction of time an average worker (64 total) spent performing useful (blue) or wasted (light gray, hatched) work, and waiting for work (light gray), normalized with respect to wall clock time. The number of batches per likelihood evaluation is 10 times greater than in Figure 5.14, and the workers spend about 10 times longer waiting for work.

when the worker is approaching the end of an assignment or alternately decides when the worker should stop its current assignment. Thus, potential improvements could come from two modifications: the master could eagerly send a `HAVE-WORK` message to a worker as soon as it recognizes that the worker is nearing the end of the assignment, and it could also combine a `HAVE-WORK` message with an `ABANDON` message. Note that in the first of these, we wouldn't want the `HAVE-WORK` messages to be sent too eagerly, since they would be based on potentially stale predictions. Another strategy for achieving better scalability would be to introduce multiple submasters, where each is responsible for managing a portion of the jobtree. We leave investigation of all these ideas as future work.

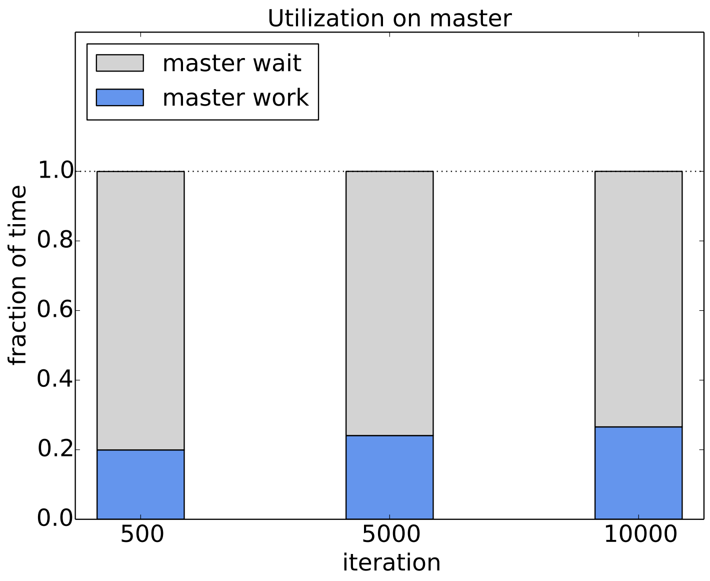


Figure 5.18: Cumulative fraction of time on master spent acting in response to messages from 64 workers (blue) and waiting for worker messages (light gray), normalized with respect to wall clock time. The number of batches per likelihood evaluation is 10 times greater than in Figure 5.15, and utilization of the master is 5 to 6 times higher.

Chapter 6

Conclusions and generalizations

We presented parallel predictive prefetching, a general framework for accelerating many widely used MCMC algorithms that are inherently serial and often slow to converge. Our approach applies to MCMC algorithms whose transition operator can be decomposed into two functions: one that produces a countable set of candidate proposal states and a second that chooses the next state from among these. Predictive prefetching uses parallel cores and speculative computation to exploit the common setting in which generating proposals is computationally fast compared to the evaluation required to choose from among them and this latter evaluation can be approximated quickly. Our first focus has been on the MH algorithm, in which predictive prefetching exploits a sequence of increasingly accurate predictors for the decision to accept or reject a proposed state. Our second focus has been on large-scale Bayesian inference, for which we identified an effective predictive model that estimates the likelihood from a subset of data. The key insight is that we model the uncertainty of these predictions with respect to the difference between the likelihood of each datum evaluated at the proposal and current state. As these evaluations are highly correlated, the variance of the differences is much smaller than the variance of the states evaluated separately, leading to significantly higher confidence in our predictions. This allows us to justify more aggressive use of parallel resources, leading to greater speedup with respect to serial execution or more naïve prefetching schemes.

The best speedup that is realistically achievable for this problem is sublinear in the number of cores but better than logarithmic, and our results achieve this. As noted in Section 3.5.3, it would be straightforward to combine predictive prefetching with parallelism at each node; we would expect this to yield much better speedups for the Bayesian inference problems we considered, which lend themselves to this kind of parallelism. Our empirical evaluation only studied Bayesian inference problems, for which we constructed fast approximations to the target density via data subsets. Other common approximations for probability distributions are formed from Taylor series expansions, *e.g.*, as used by Christen and Fox (2005), and linear or Gaussian process regressions, *e.g.*, as used by (Conrad et al., 2014). Our approach generalizes both to schemes that learn an approximation to the target density and to other MCMC algorithms with more complex structure, such as slice sampling and more sophisticated adaptive techniques.

In predictive prefetching, we maintain a tree data structure where each node corresponds to a set of parameters at which it might be useful to evaluate the target density; each node is associated with a utility. In our system, the master core represents the tree and schedules workers to the highest utility nodes. Each worker incrementally evaluates the assigned target, and each partial computation updates node utilities. Subsequently, the master might instruct workers to abandon their current work and reassign them to different nodes. The master caches partial computations at abandoned nodes and can later have other workers recommence where previous workers were stopped. Our approach is reminiscent of a recent Bayesian optimization algorithm by Swersky et al. (2014). *Bayesian optimization* alternates between proposing a set of parameters and evaluating them with respect to some potentially expensive objective function. In particular, these could be the hyperparameters to a machine learning model that take a long time to fit (Snoek et al., 2012). Swersky et al. combine a cache of 'frozen' partial evaluations, the ability to 'thaw' and continue these evaluations, a pool of new candidate parameters that haven't been evaluated at all, and an information theoretic utility model to decide what to evaluate next, *i.e.*, something frozen or something new. In this setting, all potential evaluations yield some information, but the amount of information gained depends on the evaluations that have been performed already – *e.g.*, once a particular parameter setting has been (partially) evaluated, other nearby parameter evaluations may not be expected to add much information. In contrast, in our setting, a constant but *a priori* unknown subset of potential computations must be performed; all other speculative computations are wasteful and eventually known to have zero utility. We note that the parallel Bayesian optimization strategy developed by Snoek et al. (2012), which sequentially decides what parameters to evaluate next, could be extended to incorporate the freeze-thaw framework.

An important contribution of our research has been to provide greater exposure to prefetching ideas, which did not appear to be well-known when we began. In response to our publication of a short version of this work on arXiv,[1] a statistician published a review of our work on his blog, indicating that he had previously been unfamiliar with prefetching (Robert, 2014). We are happy to report that, with colleagues, he has since combined naïve prefetching with a delayed acceptance method (Banterle et al., 2014). We hope that other researchers will also find prefetching ideas to be useful and develop more powerful predictive prefetching techniques, in particular.

Our curiosity in speculative execution is not limited to prefetching for MCMC – we are broadly interested in it as a general computational technique. In fact, this dissertation grew out of prior research that developed a computational model for exploiting speculative execution to parallelize serial programs (Waterland et al., 2013, 2014). This dissertation is a focused study of the power of speculative execution, applied to a particular class of algorithms. Our system architecture presented in Chapter 4 shares some similarities with the architecture developed in our prior work. In both, a master manages the state of computation and schedules workers to perform (speculative) computation; workers also generate

[1]This article has since been published in peer-reviewed conference proceedings (Angelino et al., 2014).

information used to form probabilistic predictions about what work to perform next. There are also significant differences; in particular, our work with MCMC makes explicit use of algorithm-level semantics and structure – this information is distilled in our central data structure, the jobtree.

Our study of MCMC in the context of speculative execution is in the spirit of a recent area of work that develops new parallel machine learning algorithms by adapting ideas from the systems community, especially database research. Most of this work focuses on optimization problems, rather than Bayesian inference. In particular, Pan et al. (2013) describe three different parallel approaches to leveraging data parallelism. When a parallel version of a serial algorithm enforces *serializability*, it maintains a strict but partial order on operations to yield output equivalent to serial execution; the partial order specifies groups of operations that may run *concurrently* (in parallel). The first method, *mutual exclusion*, maintains serializability via locks. It limits the amount of achievable parallelism and incurs potentially significant overhead due to locking, but straightforwardly maintains properties of the original algorithm, *e.g.*, correctness, if applicable. Alternatively, a *coordination-free* approach throws away locks, and with them, their associated overheads as well as the automatic retention of desirable algorithmic properties. Recht et al. (2011) applied this idea to stochastic gradient descent, rebranding it as "hogwild," and developed theoretical tools to prove its correctness under certain conditions. Both the name and general approach have gained popularity in the machine learning community. A third method, *optimistic concurrency control* (OCC), guarantees serializability while remaining lock-free. Developed by Kung and Robinson (1981), OCC proceeds similarly to the coordination-free approach, but it checks for actions that violate serializability constraints and must correct for any such actions. Machine learning algorithms that have only weak dependencies between computations on different (groups of) data items can be good candidates for coordination-free or OCC approaches. Pan et al. implement a policy that is inspired by OCC; using knowledge about specific serial machine learning algorithms, they develop concurrency control mechanisms that preserve algorithm semantics. For example, a clustering algorithm updates a global variable indicating the cluster centers. In the serial algorithm, these are always up-to-date. In their algorithm, the data are partitioned across machines, each of which maintains a possibly out-of-date, or *stale*, version of the global variable. No constraints are violated unless this variable changes in a way that affects computations on machines that don't yet know about the change, *e.g.*, when a new cluster center is introduced. When this happens, a special master core discards computations in conflict with required constraints and ensures that the correct computations are performed. Ultimately, Pan et al. suggest that we might be able to develop a continuum of concurrency policies that trade-off between correctness and speed.

To recapitulate, speculative execution is a general approach for accelerating computation by optimistically performing computation that might be useful. We view the original form of OCC as similar to a restricted form of speculative execution where the optimistic computations are based on a possibly stale understanding of the true state and pursued in a depth-first manner. In our research, we drive speculative scheduling decisions by actively predicting what computations to do and furthermore coherently qualify our predictions within a Bayesian probabilistic framework. Thus far, we have limited ourselves to speculative techniques that yield output invariant to the number of parallel cores. We agree with Pan et al. (2013) that it could be fruitful to relax hard serializability constraints, especially for machine learning algorithms, as "we may be able to partially or *probabilistically* accept non-serializable operations in a way that preserves underlying algorithm invariants." A complementary perspective suggests that areas of approximate computation or heuristic algorithms might tolerate more aggressive forms of speculative execution. Beyond machine learning algorithms, differential equation solvers present an intriguing area for further study. These computational workhorses perform forward numerical integration of systems of differential equations – an inherently serial procedure. Schober et al. (2014) recently developed a probabilistic ordinary differential equation solver that could be a good candidate for a prediction-based speculative execution framework and furthermore suggests trade-offs between accuracy and speed.

Many computational problems, especially in but not limited to machine learning, may benefit from being revisited with the arsenal of techniques from the systems community. Simultaneously, many existing systems ideas may be augmented by viewing them through the principled twin lenses of machine learning and information theory. We speculate that these complementary approaches will yield novel and useful algorithms more fully capable of exploiting future computational resources.

Bibliography

S. Ahn, A. K. Balan, and M. Welling. Bayesian posterior sampling via stochastic gradient Fisher scoring. In *Proceedings of the 29th International Conference on Machine Learning*, ICML '12, 2012.

C. Amador-Bedolla, R. Olivares-Amaya, J. Hachmann, and A. Aspuru-Guzik. Towards materials informatics for organic photovoltaics. In K. Rajan, editor, *Informatics for Materials Science and Engineering*. Elsevier, Amsterdam, 2013.

C. Andrieu and E. Moulines. On the ergodicity properties of some adaptive MCMC algorithms. *The Annals of Applied Probability*, 16(3):1462–1505, 2006.

C. Andrieu and J. Thoms. A tutorial on adaptive MCMC. *Statistics and Computing*, 18(4): 343–373, 2008.

E. Angelino, E. Kohler, A. Waterland, M. Seltzer, and R. P. Adams. Accelerating MCMC via parallel predictive prefetching. In *30th Conference on Uncertainty in Artificial Intelligence*, UAI '14, 2014.

M. Banterle, C. Grazian, and C. P. Robert. Accelerating Metropolis-Hastings algorithms: Delayed acceptance with prefetching. June 2014. Available at arXiv:1406.2660.

R. Bardenet, A. Doucet, and C. Holmes. Towards scaling up Markov chain Monte Carlo: An adaptive subsampling approach. In *Proceedings of the 31st International Conference on Machine Learning*, ICML '14, 2014.

A. E. Brockwell. Parallel Markov chain Monte Carlo simulation by pre-fetching. *Journal of Computational and Graphical Statistics*, 15(1):246–261, March 2006.

J. M. R. Byrd, S. A. Jarvis, and A. H. Bhalerao. Reducing the run-time of MCMC programs by multithreading on SMP architectures. In *IPDPS*, pages 1–8. IEEE, 2008.

J. M. R. Byrd, S. A. Jarvis, and A. H. Bhalerao. On the parallelisation of MCMC by speculative chain execution. In *IPDPS Workshops*, pages 1–8. IEEE, 2010.

J. A. Christen and C. Fox. Markov chain Monte Carlo using an approximation. *Journal of Computational and Graphical Statistics*, 14(4):795–810, 2005.

P. R. Conrad, Y. M. Marzouk, N. S. Pillai, and A. Smith. Asymptotically exact MCMC algorithms via local approximations of computationally intensive models. Feb. 2014. Available at arXiv:1402.1694.

J. Dennis and R. B. Schnabel. *Numerical methods for unconstrained optimization and nonlinear equations*. Prentice-Hall Series in Computational Mathematics, 1983.

P. Diaconis. The Markov chain Monte Carlo revolution. *Bulletin of the American Mathematical Society*, (2):179–205, Nov. 2008.

P. Diaconis, S. Holmes, and R. M. Neal. Analysis of a non-reversible Markov chain sampler. *The Annals of Applied Probability*, 10(3):726–752, 08 2000.

A. Doucet, M. Pitt, R. Kohn, and G. Deligiannidis. Efficient implementation of Markov chain Monte Carlo when using an unbiased likelihood estimator. 2014. Available at arXiv:1210.1871.

S. Duane, A. D. Kennedy, B. J. Pendleton, and D. Roweth. Hybrid Monte Carlo. *Physics Letters B*, 195(2):216–222, 1987.

D. Foreman-Mackey, D. W. Hogg, D. Lang, and J. Goodman. emcee: The MCMC hammer. 2012. Available at arXiv:1202.3665.

A. Gelman and D. B. Rubin. Inference from iterative simulation using multiple sequences. *Statistical Science*, pages 457–472, 1992.

A. Gelman, J. B. Carlin, H. S. Stern, and D. B. Rubin. *Bayesian Data Analysis, Second Edition*. Chapman and Hall/CRC, July 2003.

C. J. Geyer and A. Mira. On non-reversible Markov chains. In *Institute Communications, Volume 26: Monte Carlo Methods*, pages 93–108. American Mathematical Society, 2000.

J. Goodman and J. Weare. Ensemble samplers with affine invariance. *Communications in Applied Mathematics and Computational Science*, 5(1):65–80, 2010.

P. J. Green and A. Mira. Delayed rejection in reversible jump Metropolis-Hastings. *Biometrika*, 88(4):pp. 1035–1053, 2001.

J. Hachmann, R. Olivares-Amaya, S. Atahan-Evrenk, C. Amador-Bedolla, R. S. Sánchez-Carrera, A. Gold-Parker, L. Vogt, A. M. Brockway, and A. Aspuru-Guzik. The Harvard Clean Energy Project: Large-scale computational screening and design of organic photovoltaics on the world community grid. *The Journal of Physical Chemistry Letters*, 2(17): 2241–2251, 2011.

J. Hachmann, R. Olivares-Amaya, A. Jinich, A. L. Appleton, M. A. Blood-Forsythe, L. R. Seress, C. Román-Salgado, K. Trepte, S. Atahan-Evrenk, S. Er, S. Shrestha, R. Mondal, A. Sokolov, Z. Bao, and A. Aspuru-Guzik. Lead candidates for high-performance organic photovoltaics from high-throughput quantum chemistry - the Harvard Clean Energy Project. *Energy Environ. Sci.*, 7:698–704, 2014.

H. Haramoto, M. Matsumoto, and P. L'Ecuyer. A fast jump ahead algorithm for linear recurrences in a polynomial space. In *Proceedings of the 5th International Conference on Sequences and Their Applications*, SETA '08, pages 290–298, Berlin, Heidelberg, 2008a. Springer-Verlag.

H. Haramoto, M. Matsumoto, T. Nishimura, F. Panneton, and P. L'Ecuyer. Efficient jump ahead for F2-linear random number generators. *INFORMS J. on Computing*, 20(3):385–390, July 2008b.

W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, Apr. 1970.

M. D. Hoffman, D. M. Blei, C. Wang, and J. Paisley. Stochastic variational inference. *Journal of Machine Learning Research*, 14(1):1303–1347, May 2013.

Y. Iba. Extended ensemble Monte Carlo. *International Journal of Modern Physics C*, 12 (05):623–656, 2001.

A. Korattikara, Y. Chen, and M. Welling. Austerity in MCMC land: Cutting the Metropolis-Hastings budget. In *Proceedings of the 31st International Conference on Machine Learning*, ICML '14, 2014.

S. Kullback and R. A. Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 03 1951.

H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981. ISSN 0362-5915.

J. S. Liu, F. Liang, and W. H. Wong. The multiple-try method and local optimization in Metropolis sampling. *Journal of the American Statistical Association*, 95(449):121–134, 2000.

D. Maclaurin and R. P. Adams. Firefly Monte Carlo: Exact MCMC with subsets of data. In *Proceedings of 30th Conference on Uncertainty in Artificial Intelligence*, UAI '14, 2014.

N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21 (6):1087–1092, 1953.

S. P. Meyn and R. L. Tweedie. *Markov Chains and Stochastic Stability.* Communications and Control Engineering Series. Springer-Verlag London Ltd., London, 1993.

N. Murata. *A Statistical Study on On-line Learning.* Cambridge University Press, Cambridge, UK, 1998.

I. Murray. *Advances in Markov chain Monte Carlo methods.* PhD thesis, Gatsby Computational Neuroscience Unit, University College London, 2007.

R. M. Neal. Slice sampling. *The Annals of Statistics*, 31:705–767, 2003.

R. M. Neal. Improving asymptotic variance of MCMC estimators: Non-reversible chains are better. Technical Report 0406, University of Toronto, 2004.

R. M. Neal. MCMC using Hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, 54:113–162, 2010.

R. M. Neal. How to view an MCMC simulation as a permutation, with applications to parallel simulation and improved importance sampling. Technical Report 1201, Dept. of Statistics, University of Toronto, 2012.

W. Neiswanger, C. Wang, and E. Xing. Asymptotically exact, embarrassingly parallel MCMC. In *30th Conference on Uncertainty in Artificial Intelligence*, UAI '14, 2014.

R. Nishihara, I. Murray, and R. P. Adams. Parallel MCMC with generalized elliptical slice sampling. *Journal of Machine Learning Research*, 15:2087–2112, 2014.

R. Olivares-Amaya, C. Amador-Bedolla, J. Hachmann, S. Atahan-Evrenk, R. S. Sánchez-Carrera, L. Vogt, and A. Aspuru-Guzik. Accelerated computational discovery of high-performance materials for organic photovoltaics by means of cheminformatics. *Energy Environ. Sci.*, 4:4849–4861, 2011.

X. Pan, J. E. Gonzalez, S. Jegelka, T. Broderick, and M. I. Jordan. Optimistic concurrency control for distributed unsupervised learning. In *Advances in Neural Information Processing Systems 26*, NIPS '13, pages 1403–1411, 2013.

T. Park and G. Casella. The Bayesian Lasso. *Journal of the American Statistical Association*, 103(482):681–686, 2008.

J. G. Propp and D. B. Wilson. Exact sampling with coupled Markov chains and applications to statistical mechanics. *Random Structures and Algorithms*, 9(1&2):223–252, 1996.

B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems 23*, NIPS '10, pages 693–701, 2011.

C. Robert. Accelerating MCMC via parallel predictive prefetching, Apr. 2014. URL `http://xianblog.wordpress.com/`.

G. O. Roberts, A. Gelman, and W. R. Gilks. Weak convergence and optimal scaling of random walk Metropolis algorithms. *Annals of Applied Probability*, 7:110–120, 1997.

M. Schober, D. Duvenaud, and P. Hennig. Probabilistic ODE solvers with Runge-Kutta means. 2014. Available at arXiv:1406.2582.

S. L. Scott, A. W. Blocker, and F. V. Bonassi. Bayes and big data: The consensus Monte Carlo algorithm. In *Bayes 250*, 2013.

J. Snoek, H. Larochelle, and R. P. Adams. Practical Bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems 25*, NIPS '12, pages 2951–2959, 2012.

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

Y. Sun, F. J. Gomez, and J. Schmidhuber. Improving the asymptotic performance of Markov chain Monte-Carlo by inserting vortices. In *Advances in Neural Information Processing Systems 23*, NIPS '10, pages 2235–2243, 2010.

K. Swersky, J. Snoek, and R. P. Adams. Freeze-thaw Bayesian optimization. June 2014. Available at arXiv:1406.3896.

R. Tibshirani. Regression shrinkage and selection via the Lasso. *Journal of the Royal Statistical Society, Series B*, 58:267–288, 1994.

L. Tierney and A. Mira. Some adaptive Monte Carlo methods for Bayesian inference. *Statistics in Medicine*, 18:2507–2515, 1999.

M. Tingley. *Towards the Quantum Machine: Using Scalable Machine Learning Methods to Predict Photovoltaic Efficacy of Organic Molecules.* Undergraduate thesis, Harvard College, 2014.

J. von Neumann. Various techniques used in connection with random digits. *Journal of Research of the National Bureau of Standards. Applied Mathematics Series*, 12:36–38, 1951.

X. Wang and D. B. Dunson. Parallel MCMC via Weierstrass sampler. 2013. Available at arXiv:1312.4605.

A. Waterland, E. Angelino, E. D. Cubuk, E. Kaxiras, R. P. Adams, J. Appavoo, and M. Seltzer. Computational caches. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 8:1–8:7, New York, NY, USA, 2013. ACM.

A. Waterland, E. Angelino, R. P. Adams, J. Appavoo, and M. Seltzer. ASC: Automatically scalable computation. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 575–590, New York, NY, USA, 2014. ACM.

M. Welling and Y. W. Teh. Bayesian learning via stochastic gradient Langevin dynamics. In *Proceedings of the 28th International Conference on Machine Learning*, ICML '11, 2011.

E. Witte, R. Chamberlain, and M. Franklin. Parallel simulated annealing using speculative computation. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):483–494, 1991.

Appendix A

Additional parameters in experiments

The mixture of eight, eight-dimensional Gaussians follows that by Nishihara et al. (2014). The ϕ_k values[1] are set to

$$\begin{pmatrix}\phi_1\\ \phi_2\\ \phi_3\\ \phi_4\\ \phi_5\\ \phi_6\\ \phi_7\\ \phi_8\end{pmatrix} = \begin{pmatrix}0.2456 & 0.8211 & 0.3065 & 0.9171 & 0.9674 & 0.5055 & 0.535 & 0.7781\\ 0.1852 & 0.774 & 0.9248 & 0.8285 & 0.7948 & 0.460 & 0.9904 & 0.6430\\ 0.7135 & 0.8969 & 0.7882 & 0.7179 & 0.8707 & 0.1549 & 0.364 & 0.7309\\ 0.3507 & 0.8099 & 0.0669 & 0.2366 & 0.7635 & 0.5878 & 0.5188 & 0.7846\\ 0.186 & 0.3913 & 0.7746 & 0.3846 & 0.1483 & 0.4110 & 0.5936 & 0.5528\\ 0.2550 & 0.7924 & 0.5779 & 0.5291 & 0.2643 & 0.7684 & 0.3859 & 0.9556\\ 0.3698 & 0.1247 & 0.1504 & 0.8657 & 0.9061 & 0.2281 & 0.9170 & 0.9552\\ 0.354 & 0.3176 & 0.2076 & 0.0267 & 0.6507 & 0.0931 & 0.2434 & 0.2387\end{pmatrix}.$$

[1]Personal communication with Robert Nishihara.