ACCELERATION OF SEQUENTIAL MONTE CARLO METHODS VIA PARALLELIZATION OF RESAMPLING ALGORITHMS

by

Hakan Güldaş

B.S., Mathematics, Bilkent University, 2011

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2015

ACKNOWLEDGEMENTS

I would like to thank to my supervisor Assoc. Prof. Ali Taylan Cemgil for all his support and guidance throughout the course of this work. His attention and enthusiasm has been a great motivation for me in this work and he has been an outstanding mentor through my academic endeavours. I also would like to thank to my examiners Prof. Can Özturan and Assoc. Prof. Atilla Yılmaz, for their valuable feedback on my thesis.

I would like to thank to our collaborators from the University of Bristol, Nick Whiteley and Kari Heine, for their invaluable contributions to my work and for hosting me in Bristol as part of this collaboration. I also would like to thank to the members of the Perceptual Intelligence Laboratory for their help, support and friendship. Particularly, many thanks to Ömer Deniz Akyıldız for the fruitful discussions we had.

Finally, I would like to thank my family and to Müzeyyen for everything I have and I succeeded, no words can describe your value to me.

ABSTRACT

ACCELERATION OF SEQUENTIAL MONTE CARLO METHODS VIA PARALLELIZATION OF RESAMPLING ALGORITHMS

Sequential Monte Carlo (SMC) methods, also known as particle filters, are a popular set of tools in Bayesian inference on non-linear non-Gaussian state space models. While these algorithms are traditionally developed with serial computation in mind, recent developments in the field of parallel computation has caught the attention of SMC community as well and there have been efforts to parallelize particle filters. Efforts on parallelization of particle filters are focused on the parallelization of resampling algorithms. In this thesis, we investigate the parallelization of resampling algorithms on massively parallel architectures. We present implementations of classical resampling algorithms on graphical processing units (GPU) and give an asymptotic analysis of their computation time. We present a recent framework called *augmented resampling* that can be used to produce resampling algorithms specifically designed to work on some parallel computing architecture. Within this framework, we implement the *butterfly resampling* algorithms, that works under limited degree of communication between computational units, on GPUs and present asymptotic analysis of their computation time. We present theoretical results on convergence properties of butterfly resampling algorithms and conduct simulations to verify these theoretical results, to obtain practical guidelines for their implementation on GPUs and to compare their performance to classical resampling algorithms. We see that butterfly multinomial resampling algorithm can provide upto six times speed-up over classical multinomial resampling algorithm, while keeping the Monte Carlo error at a competitive level.

ÖZET

YENİDEN ÖRNEKLEME ALGORİTMALARININ PARALELLEŞTİRİLMESİ YOLUYLA ARDIŞIK MONTE CARLO YÖNTEMLERİNİN HIZLANDIRILMASI

Parçacık süzgeçleri olarak da bilinen ardışık Monte Carlo (AMC) yöntemleri, lineer ve Gaussian olmayan durum uzaylarında Bayesçi kestirim için çokça kullanılan bir araçlar bütünüdür. Bu yöntemler geleneksel olarak seri hesaplama mimarileri düşünülerek geliştirilmiş olsa da, paralel hesaplama konusundaki son gelişmeler AMC camiasının da dikkatini çekmiş ve parçacık süzgeçlerinin paralelleştirilmesine yönelik çalışmalar olmuştur. Parçacık süzgeçlerinin paralelleştirilmesi yönündeki çabalar yeniden örnekleme algoritmalarının paralelleştirilmesi üzerine yoğunlaşmıştır. Bu tezde, yeniden örnekleme algoritmalarının, büyük çapta paralel mimariler üzerinde gerçeklenmesini inceliyoruz. Klasik yeniden örnekleme algoritmalarının grafik işleme üniteleri (GPU) üzerinde gerçeklenmelerinin yanı sıra hesaplama zamanı maliyetlerini analiz ediyoruz. Klasik yeniden örnekleme algoritmalarına ilaveten son zamanlarda öne sürülmüş olan ve eldeki paralel mimariye özgün yeniden hesaplama algoritmaları tasarlanmasına olanak sağlayan *genişletilmiş yeniden örnekleme* çerçevesini sunuyoruz. Bu çerçeve içinde geliştirilmiş olan ve kısıtlı iletişim koşulları altında çalışabilen *kelebek yeniden örnekleme* algoritmasını gerçekleyip hesaplama zamanı analizini sunuyoruz. Kelebek yeniden algoritmasının yakınsaklığına ilişkin teorik sonuçları sunup bu sonuçları doğrulamak, GPU üzerinde gerçeklenmesine dair rehber ilkeler belirlemek ve klasik yeniden örnekleme algoritmalarıyla karşılaştırmak için deneyler düzenliyoruz. Bu deneyler sonucunda görüyoruz ki, kelebek yeniden örnekleme algoritmaları klasik algoritmalardan altı kat kadar hızlı olabilirken Monte Carlo hatasını rekabetçi bir seviyede tutabilir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

$\mathbf{1}_N$	$N \times N$ matrix with all elements equal to $1/N$
$a = (a^1, \ldots, a^N)$	A vector of length N
$a_{0:n} = (a_0, a_1, \ldots, a_n)$	A sequence of length n
$f(x, \cdot), g(x, \cdot)$	Markov transition kernels conditioned on x
$g_n(x)$	Observation likelihood at time n at state x
I_N	$N \times N$ identity matrix
$Q(\cdot \mid X_{0:n})$	Proposition kernel of particle filter
$X = \{X_n\}_{n \geq 0}$	A stochastic process
$(\mathsf{X}, \mathcal{X})$	A state space X equipped with Borel σ-algebra $\mathcal{X}$
$\frac{d\mu}{d\nu}(\cdot)$	Radon-Nikodym derivative of μ w.r.t. ν
$\mu(\cdot)$	Probability measure
$(\Omega, \mathcal{F}, \mathbb{P})$	The probability space on which all random variables are defined

LIST OF ACRONYMS/ABBREVIATIONS

AMSE	Averaged Mean Squared Errors
ARPF	Adaptive Resampling Particle Filter
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
EM	Expectation-Maximization
ESS	Effective Sample Size
GPU	Graphical Processing Unit
HMM	Hidden Markov Model
MSE	Mean Squared Error
LMSE	Local Mean Squared Error
PF	Particle Filter
SIMT	Single Instruction Multiple Threads
SIS	Sequential Importance Sampling
SIR	Sequential Importance Resampling
SMC	Sequential Monte Carlo

1. INTRODUCTION

Estimating the current state of an uncertain system that changes over time has been an important problem in many diverse fields ranging from engineering to econometrics, from ocean and atmosphere sciences to chemometrics. It is often the case that we have a stochastically changing system and we have only partial information about the system through noisy observations. Bayesian treatment of this problem involves computation of posterior distributions of *hidden* states of the system or expectations under these distributions. When underlying stochastic dynamics involves only linear functions of Gaussian random variables, posterior distributions are also Gaussian and mean and covariances of these distributions can be easily computed by using recursive equations of celebrated Kalman filter [1, 2]. However, when we remove either of the linearity or Gaussianity assumptions, we seldom have a tractable model in which exact computation can be performed given reasonable amount of computational resources. In this case, we need to resort to approximate methods such as extended Kalman filter and unscented Kalman filter [3,4] or sequential Monte Carlo methods [5–9], also known as particle filters (PF) which is the main topic of this thesis.

More formally, we have a stochastic process $(X, Y) = \{(X_n, Y_n)\}_{n\geq 0}$ defined on a probability space $(\Omega, \mathcal{F}, \mathbb{P})$, taking values on the product measure space $(\mathsf{X} \times \mathsf{Y}, \mathcal{X} \otimes \mathcal{Y})$, where $X = \{X_n\}_{t\geq 0}$ is taking values on $(\mathsf{X}, \mathcal{X})$, typically $(\mathsf{X}, \mathcal{X}) = (\mathbb{R}^d, \mathcal{B}(\mathbb{R}^d))$, the d-dimensional real vector space equipped with Borel σ-algebra, called the hidden part, and $Y = \{Y_n\}_{t\geq 0}$ is taking values on $(\mathsf{Y}, \mathcal{Y})$, typically $(\mathsf{Y}, \mathcal{Y}) = (\mathbb{R}^m, \mathcal{B}(\mathbb{R}^m))$, the m-dimensional real vector space equipped with Borel σ-algebra, called the observable part. We are interested in the filtering distributions,

$$\hat{\pi}_n(A) \coloneqq \mathbb{P}(X_n \in A | \{Y_{0:n} = y_{0:n}\}) \text{ for } A \in \mathcal{X}, n \geq 0, \tag{1.1}$$

one-step ahead predictive distributions

$$\pi_n(A) \coloneqq \mathbb{P}(X_n \in A | \{Y_{0:n-1} = y_{0:n-1}\}) \text{ for } A \in \mathcal{X}, n \geq 1, \tag{1.2}$$

and/or expectations under these distribution. Amongst the most common settings for filtering problem is the hidden Markov model (HMM), in which the underlying probability model is specified as follows :

$$\begin{aligned} X_0 &\sim \mu, \\ X_n \mid \{X_{n-1} = x_{n-1}\} &\sim f(x_{n-1}, \cdot), \\ Y_n \mid \{X_n \ = \ x_n\} &\sim g(x_n, \cdot), \end{aligned} \tag{1.3}$$

where μ is a probability measure on $(\mathsf{X}, \mathcal{X})$, f is the time homogeneous Markov transition kernel from $(\mathsf{X}, \mathcal{X})$ into itself and g is the time homogeneous Markov transition kernel from $(\mathsf{X}, \mathcal{X})$ to $(\mathsf{Y}, \mathcal{Y})$ which is absolutely continuous with respect to some reference measure dy. Here we fix an observation sequence $y = \{y_0, y_1, \dots\}$ once and for all and denote $g_n(x)$ the likelihood of x at point y_n, $g(x, dy_n)$ for all $n \geq 0$. Within HMM setting, filtering distributions, $\hat{\pi}_n$, and predictive distributions π_n satisfy following recursive equations:

$$\begin{aligned} \pi_0(dx_0) = \mu(dx_0), \qquad \hat{\pi}_0(dx_0) &= \frac{g_0(x_0)\pi_0(dx_0)}{\int g_0(x_0)\pi_0(dx_0)}, \\ \pi_n(dx_n) &= \int f(x_{n-1}, dx_n)\hat{\pi}_{n-1}(dx_{n-1}), \\ \hat{\pi}_n(dx_n) &= \frac{g_n(x_n)\pi_n(dx_n)}{\int g_n(x_n)\pi_n(dx_n)}, \end{aligned} \tag{1.4}$$

Other related quantities of interest are the smoothing distribution,

$$p_n(A) = \mathbb{P}(X_{0:n} \in A | \{Y_{0:n} = y_{0:n}\}) \text{ for } A \in \mathcal{X}^{\otimes(n+1)},$$

and marginal likelihood of first n observations, $Z_n = \mathbb{P}(Y_{0:n} \in dy_{0:n})$. These quantities are encountered very frequently in the practice and they satisfy the following set of

equations:

$$\pi_n(dx_n) = \int f(x_{n-1}, dx_n) p_{n-1}(dx_{0:n-1}),$$
$$\hat{\pi}_n(dx_n) = \int p_n(dx_{0:n-1}),$$
$$Z_0 = \int g_0(x_0)\pi_0(dx_0),$$
$$Z_n = Z_{n-1} \int g_{n-1}(x_n)\pi_{n-1}(dx_n).$$

Particle filtering algorithms, which will be described in Section 1.2, have been a popular set of tools in Bayesian inference since their first instances [10, 11] were introduced in 1990s. While there have been many improvements to the original algorithm each targeting a different shortcoming [12–17], statistical properties of particle filters such as stability, consistency of estimators and asymptotics of estimation errors have been studied as well [18–25]. Now, the mathematical theory behind particle filtering is better understood, although far from being complete, and we are much aware of their potentials and limitations. On the other hand, until very recently, when developing these algorithms very little attention was paid to the computational architecture on which they are implemented with the presumption that they would be implemented with sequential processing. However, with the recent advances in processor technology parallel computation came into prominence and these advances necessitate the development of particle filtering algorithms better suited to parallel computing platforms.

1.1. Recent Developments in Parallel Computation

The concept of parallel computation has been around for decades. Parallelism in hardware level is achieved generally by attaching several CPUs and memory units together to a common bus or by a collection of stand-alone computer systems connected by a network [26, Chapter 7]. Parallelism in software level is achieved by libraries and APIs. In multiprocessor systems, where a small number of processors communicate through a shared memory space, common software platforms include POSIX threads

and OpenMP APIs. In distributed systems, where processors communicate through a network, the most common software platform is the industry standard Message Passing Interface (MPI) [27] which has several implementations. Although parallel programming was employed mainly for high performance computing in the past, the trend of increasing the computation capability of computers by designing faster processor is replaced by the trend of designing parallel computing architectures in the form of multi-core processors in the last decade [28]. This is due to the physical contraints on the power-frequency scaling, the so-called *power wall* phenomenon.

Further development in the realm of parallel computation came with the modern graphical processing unit (GPU) architectures. Although GPUs were originally designed to accomodate graphical computations, which are more prone to massively parallel pipelines in hardware level, recent developments in GPU technology allow us to perform massively parallel general purpose computations on commodity hardware [29]. An early attempt of general purpose GPU programming language is BrookGPU by Stanford University graphics group [30]. With the introduction of CUDA [31] and OpenCL [32] frameworks, software developers have more access to GPU resources for general purpose computation and there has been a substantial accumulation of literature on the subject of general purpose GPU computing [29,33,34]. These developments caught the attention of Monte Carlo research community as well and there has been a substantial output on the topic in recent years [35–39]. In line with this trend, in this thesis we investigate the problem of parallelization of particle filters via parallel resampling algorithms.

1.2. Particle Filters and Prior Work on Their Parallelization

The key idea of particle filters is to use empirical distributions of weighted samples, termed particles, to estimate the filtering and predictive distributions. More formally, for each time step $n = 0, \ldots,$ we have random vectors of particles $\xi_n = (\xi_n^1, \ldots, \xi_n^N), \hat{\xi}_n = (\hat{\xi}_n^1, \ldots, \hat{\xi}_n^N) \in \mathsf{X}^N$ and corresponding vectors of nonnegative weights $w_n = (w_n^1, \ldots, w_n^N), \hat{w}_n = (\hat{w}_n^1, \ldots, \hat{w}_n^N) \in \mathbb{R}_+^N$. We approximate the predictive distribution π_n, filtering distribution $\hat{\pi}_n$, and smoothing distribution p_n by random empirical

measures

$$\pi_n^N(dx) = \sum_{i=1}^{N} \frac{w_n^i}{\sum_j w_n^j} \delta_{\xi_n^i}(dx), \tag{1.5}$$

$$\hat{\pi}_n^N(dx) = \sum_{i=1}^{N} \frac{\hat{w}_n^i}{\sum_j \hat{w}_n^j} \delta_{\hat{\xi}_n^i}(dx), \tag{1.6}$$

$$p_n^N(dx_{0:n}) = \sum_{i=1}^{N} \frac{\hat{w}_n^i}{\sum_j \hat{w}_n^j} \delta_{\hat{\xi}_{0:n}^i}(dx_{0:n}), \tag{1.7}$$

respectively and for a test function $\varphi : \mathsf{X} \to \mathbb{R}$, we approximate the expectation under predictive distibution $\pi_n(\varphi)$ and filtering distribution $\hat{\pi}_n(\varphi)$ by the empirical means

$$\pi_n^N(\varphi) = \sum_{i=1}^{N} \frac{w_n^i}{\sum_j w_n^j} \varphi(\xi_n^i), \tag{1.8}$$

$$\hat{\pi}_n^N(\varphi) = \sum_{i=1}^{N} \frac{\hat{w}_n^i}{\sum_j \hat{w}_n^j} \varphi(\hat{\xi}_n^i), \tag{1.9}$$

where we have $w_n = \hat{w}_{n-1}$ and $w_0 \equiv 1$.

Before we move to the generic case, we give two introductory instances of the particle filtering algorithms: the prototypical sequential importance sampling algorithm (SIS) [40] and first practical particle filter algorithm, sequential importance resampling [10]. SIS algorithm constitute the basis of the sequential Monte Carlo methods developed in last decades [6]. As the name suggests, SIS is an extension of importance sampling procedure to the sequences of distributions. Recall that, in importance sampling, to approximate a distribution μ from which it is hard to produce samples or expectations under this distribution of a function φ, $\mu(\varphi) = \int \varphi(x)\mu(dx)$, we sample from another distribution ν from which it is easier to generate samples and which dominates μ and use these samples weighted with Radon-Nikodym derivatives evaluated at those points:

$$\mu(\varphi) \approx \frac{1}{N} \sum_{i=1}^{N} \frac{d\mu}{d\nu}(\xi^i)\varphi(\xi^i), \text{ where } \xi \overset{\text{i.i.d.}}{\sim} \nu.$$

It is easy to show that this estimator is an unbiased estimator of $\mu(\varphi)$. When we cannot evaluate the Radon-Nikodym derivative exactly but can compute it upto a positive constant α, i.e. we can compute $\alpha\frac{d\mu}{d\nu}$, we can use the normalized weights as follows:

$$\mu(\varphi) \approx \sum_{i=1}^{N} \frac{w^i}{\sum_{j=1}^{N} w^j}\varphi(\xi^i), \text{ where } \xi \overset{\text{i.i.d.}}{\sim} \nu \text{ and } w^i = \alpha\frac{d\mu}{d\nu}(\xi^i).$$

This is no longer an unbiased estimator of $\mu(\varphi)$ because of the normalization step, however it can be shown that this is a consistent estimator, i.e. as we let $N \to \infty$ this sum will approach to $\mu(\varphi)$. This scheme suggests the idea of approximating μ with the weighted empricial measure:

$$\mu(\cdot) \approx \mu^N(\cdot) = \sum_{i=1}^{N} \frac{w^i}{\sum_{j=1}^{N} w^j}\delta_{\xi}^{i}(\cdot), \text{ where } \xi \overset{\text{i.i.d.}}{\sim} \nu \text{ and } w^i = \alpha\frac{d\mu}{d\nu}(\xi^i).$$

SIS algorithm applies the importance sampling procedure to the predictive and filtering distributions given in 1.4. Assume we have the underlying probability model given in 1.3 and suppose we can generate samples from a family of proposal distributions $\{Q_n(\cdot|\xi_{0:n-1})\}_{n\geq 1}$ which dominate $f(\xi_{n-1}, \cdot)$ and which may also depend on the observation sequence $y = \{y_0, y_1, \dots\}$ but we suppress this in notation since the observation sequence is fixed and let $q_n = Q_n Q_{n-1} \dots \mu$. SIS proceeds by recursively propagating samples $\xi_{0:n}^i$ and weighing these samples with w_n^i by using the following equation:

$$w_n^i = \frac{dp_n}{dq_n}(\xi_{0:n}^i) = \frac{g_n(\xi_n^i)f(\xi_{n-1}^i, d\xi_n^i)}{Q_n(d\xi_n^i|\xi_{0:n-1}^i)}\frac{dp_{n-1}}{dq_{n-1}}(\xi_{0:n-1}^i) = \frac{g_n(\xi_n^i)f(\xi_{n-1}^i, d\xi_n^i)}{Q_n(d\xi_n^i|\xi_{0:n-1}^i)}w_{n-1}^i, \quad (1.10)$$

where the term $\dfrac{g_n(\xi_n^i)f(\xi_{n-1}^i, d\xi_n^i)}{Q_n(d\xi_n^i|\xi_{0:n-1}^i)} = v_n^i$ is called the incremental weight. Then by

marginalization, we get the empirical predictive,filtering and smoothing distributions

$$\pi_n^N(\cdot) = \sum_{i=1}^{N} \frac{w_{n-1}^i}{\sum_j w_{n-1}^j} \delta_{\xi_n^i}(\cdot),$$
$$\hat{\pi}_n^N(\cdot) = \sum_{i=1}^{N} \frac{w_n^i}{\sum_j w_n^j} \delta_{\xi_n^i}(\cdot),$$
$$p_n^N(\cdot) = \sum_{i=1}^{N} \frac{w_n^i}{\sum_j w_n^j} \delta_{\xi_{0:n}^i}(\cdot)$$

We give the pseudocode for SIS in Figure 1.1.

function SEQUENTIALIMPORTANCESAMPLING
  for $n = 0, \ldots$ **do**
    for each $i \in \{1, \ldots, N\}$ **do**
      $\xi_n^i \sim Q_n(\cdot|\xi_{0:n-1}^i)$ //Sample particles
      $w_n^i \leftarrow \frac{g_n(\xi_n^i) f(\xi_{n-1}^i, d\xi_n^i)}{q_n(d\xi_n^i|\xi_{0:n-1}^i)} w_{n-1}^i$ //Compute weights
    end for
  end for
end function

Figure 1.1. Sequential Importance Sampling.

SIS algorithm is *embarrasingly parallel*, meaning it does not need any effort to separate the algorithm into parallel tasks, each particle can be propagated and weighed separately without needing any communication between them, since they are independently sampled. However, this independence structure causes the degeneracy problem, namely the problem that after a few iterations all but one of the particles have negligible normalized weights and it is shown that this situation cannot be avoided if the independence assumption is to be hold [41]. The remedy to this situation is the perform a resampling step for every time step in which particles with small weights are eliminated and particles with big weights are replicated.

SIR algorithm is the first practical particle filtering algorithm based on the above presented ideas. Its only difference from the SIS algorithm is the addition of the

resampling step. There are many possible ways of performing this resampling step but the original algorithm implements the multinomial resampling, which is simply sampling with replacement where probabilities are proportional to the weights. The pseudocode for SIR is given in Figure 1.2. Unlike sampling and weight computation parts, resampling step cannot be parallelized in a straightforward manner. To indicate this, we give the sampling and weight computation parts in a **for each** block, while resampling step is given in a **for** block, following the exposition in [42]. The resulting empirical predictive, filtering and smoothing distributions of SIR algorithm are given by:

$$\pi_n^N(\cdot) = \frac{1}{N}\sum_{i=1}^{N} \delta_{\xi_n^i}(\cdot),$$
$$\hat{\pi}_n^N(\cdot) = \frac{1}{N}\sum_{i=1}^{N} \delta_{\hat{\xi}_n^i}(\cdot),$$
$$p_n^N = \frac{1}{N}\sum_{i=1}^{N} \delta_{\hat{\xi}_{0:n}^i}(\cdot)$$

function SequentialImportanceResampling
  for $n = 0, \ldots$ **do**
    for each $i \in \{1, \ldots, N\}$ **do**
      $\xi_n^i \sim Q_n(\cdot|\hat{\xi}_{0:n-1}^i)$ //Sample particles
      $w_n^i \leftarrow \frac{g_n(\xi_n^i) f(\hat{\xi}_{n-1}^i, d\xi_n^i)}{q_n(d\xi_n^i|\hat{\xi}_{0:n-1}^i)} \hat{w}_{n-1}^i$ //Compute weights
    end for
    for $i = 0, \ldots, N$ **do**
      $\hat{\xi}_{0:n}^i \sim \sum_{i=1}^N \frac{w_n^i}{\sum_j w_n^j} \delta_{\xi_{0:n}^i}(\cdot)$ //Resample
      $\hat{w}_n^i = \frac{1}{n}\sum_j w_n^j$ //Update weights
    end for
  end for
end function

Figure 1.2. Sequential Importance Resampling.

Resampling step solves the problem that filtering distributions $\{\hat{\pi}_n^N\}_{n\geq 0}$ and predictive distributions $\{\pi_n^N\}_{n\geq 0}$ degenerates over time. Nevertheless, it leads to another problem called *sample impoverishment*. Note that in line 8 in Figure 1.2., we resample the whole path $\hat{\xi}_{0:n}^i$. This causes the initial parts of the paths to be replicated repeatedly and eventually initial parts of the whole sample paths collapse into one. The techniques to avoid the sample impoverishment problem include the use of MCMC [15] and continuous approximations of empirical distributions [17]. Many other improvements of SIR algorithm have been proposed, for a survey of particle filtering algorithms see for example [5, 6, 8, 9].

A practical extension of SIR algorithm is adaptive resampling particle filter (ARPF) algorithm, first introduced in [12]. The aim of the ARPF is to reduce the computational burden associated to the resampling operation. ARPF achieves this by performing resampling only when it is necessary, where the degree of necessity is assessed by the effective sample size (ESS), a quantity that measures the variability amongst the weights and which also shows up naturally in the study of stability properties of particle filters [43]. For a given set of weights $w = (w^i)_{i=1}^N$, ESS is defined as follows:

$$\mathcal{E}^N(w) = \frac{(\sum_{i=1}^N w^i)^2}{\sum_{i=1}^N (w^i)^2} \in [1, N]. \tag{1.11}$$

The pseudocode for ARPF algorithm that takes ESS threshold τ as an input parameter is given in Figure 1.3.

So far, in the algorithms we described, we used the multinomial resampling as our principal resampling algorithm. As we pointed out before, there are other ways of performing resampling step and the focus of this thesis will be on the parallelization of the resampling step, since other parts of the particle filter algorithms are readily parallel. For this reason we postpone the details of the existing resampling algorithms to the next chapters. To deal with the resampling part in isolation from the other parts of the particle filter algorithm, we describe a generic form of the particle filter

function AdaptiveResamplingParticleFilter(τ)
  for $n = 0, \dots$ **do**
    for each $i \in \{1, \dots, N\}$ **do**
      $\xi_n^i \sim Q_n(\cdot|\hat{\xi}_{0:n-1}^i)$ //Sample particles
      $w_n^i \leftarrow \frac{g_n(\xi_n^i) f(\hat{\xi}_{n-1}^i, d\xi_n^i)}{q_n(d\xi_n^i|\hat{\xi}_{0:n-1}^i)} \hat{w}_{n-1}^i$ //Compute weights
    end for
    if $\mathcal{E}^N(w_n) < \tau$ **then** //ESS is below threshold
      for $i = 0, \dots, N$ **do**
        $\hat{\xi}_{0:n}^i \sim \sum_{i=1}^N \frac{w_n^i}{\sum_j w_n^j} \delta_{\xi_{0:n}^i}(\cdot)$ //Resample
        $\hat{w}_n^i = \frac{1}{n} \sum_j w_n^j$ //Update weights
      end for
    else //ESS is above threshold
      $\hat{\xi}_{0:n} \leftarrow \xi_{0:n}, \hat{w}_n \leftarrow w_n$
    end if
  end for
end function

Figure 1.3. Adaptive Resampling Particle Filter.

algorithm. A generic particle filter algorithm for HMMs involves :

(i) Sampling from a sequence of proposal distributions $\{Q_n(\cdot|\xi_{0:n-1})\}_{n\geq 0}$ which dominate $f(\xi_{n-1}, \cdot)$, here we suppress the dependence on $y = (y_1, y_2, \dots)$ since it is fixed,

(ii) Weighting these samples as in importance sampling, but recursively at each time step,

(iii) Resampling according to these weights.

Pseudocode for this procedure is given in Figure 1.4.. Resampling is performed via a generic procedure that takes a particle-weight set ξ_n, w_n and returns another particle-weight set $\hat{\xi}_n, \hat{w}_n$, where for each $i = 1, \dots, N$ we have $\hat{\xi}_n^i = \xi_n^j$ for some $j = 1, \dots, N$. To illustrate the key ideas, we will focus on the relatively simple and popular bootstrap

filter, in which sampling distributions are $Q_0(dx_0) = \mu(dx_0)$ and $Q_n(dx_n|x_{1:n-1}, y_{1:n-1}) = f(x_{n-1}, dx_n), n \geq 1$ and its adaptive version. Again, to emphasize the contrast we enclose readily parallel parts of the algorithms in **for each** blocks while we enclose sequential parts into **for** blocks, following the exposition in [42].

1: **function** GENERICPARTICLEFILTER
2: **for** $n = 0, \ldots$ **do**
3: **for each** $i \in \{1, \ldots, N\}$ **do**
4: $\xi_n^i \sim Q_n(\cdot|\hat{\xi}_{0:n-1}^i)$ //Sample particles
5: $w_n^i \leftarrow \frac{g(y_n, \xi_n^i) f(\xi_n^i, \hat{\xi}_{n-1}^i)}{Q_n(\xi_n^i|\hat{\xi}_{0:n-1}^i)} \hat{w}_{n-1}^i$ //Compute weights
6: **end for**
7: $(\hat{\xi}_n, \hat{w}_n) \leftarrow$ RESAMPLE$(\xi_n, w_n,$ additional parameters$)$ //Resample
8: **end for**
9: **end function**

Figure 1.4. Generic Particle Filter.

Implementation of resampling part in parallel computing platforms has been considered before and there are different approaches to the problem in the literature. One common approach is to implement existing sequential resampling algorithms in a clever way [44, 45] or running independent particle filters in parallel and combining their results [46]. Another line of research focuses on changing the structure of particle filter so that the resulting algorithm is more prone to parallel implementation [47–49]. These works exclusively consider the implementation of particle filters on distributed architectures. Recent works on parallelization of resampling algorithms include the idea of constraining the interaction between particles when resampling [43], partitioning the particle set into blocks that interact with each other [50] and using Metropolis-Hastings or rejection sampling instead of standard resampling methods [42]. The authors of [42] particularly focus on the GPU implementations of the resampling algorithms and provide upto 30 times speed-up over serial implementations of the same algorithms on CPUs. In this thesis, we exclusively consider the augmented resampling scheme presented recently in [51, 52]. Different aspects of the augmented resampling resembles the αSMC framework of [43] and particle island model of [50] but similarities are superficial.

An important factor on the efficiency of parallel programs is the communication patterns of the computational units. Specifically, in a particle filter implementation, particles interact with each other via the communication between the computational units on which they are implemented. Since particles interact with each other at the resampling part, the communication overhead becomes the bottleneck in the resampling part. The pattern of interactions between particles, which is also closely related to the conditional independence structure of the random variables involved in the resampling, will be called *interaction structure* of the resampling algorithm. Efficient implementations of the particle filters on parallel platforms necessitate the interaction structure of the resampling mechanism to be mapped properly to the underlying communication structure of the parallel platform.

The main feature of augmented resampling is that it allows us to develop new resampling mechanisms respecting the communication structures of different parallel computing platforms. In the case of GPUs for example, this communication structure manifests itself in the access patterns to the global memory or via the barrier synchronization of threads and these features impose a local structure of threads. Within augmented resampling framework, we investigate the *butterfly resampling* algorithm that can be mapped to the local structure of the GPU architecture better than the standard resampling algorithms in the literature. Roughly speaking, butterfly resampling distributes the *dense interaction structure* of multinomial resampling into stages of *sparse interaction structure* by incrementing the overall interaction at each stage. A dense interaction structure is characterized by the heavy communication traffic between the computation units and in general it is this communication traffic that hinders the parallelization of serial algorithms.

Sparse interactions amongst particles in the butterfly resampling facilitate more parallelizable particle filter algorithms. The price to pay for this sparse interaction structure shows up in the scaling factor or in the variance of asymptotic approximation errors. The scaling factor of the asymptotic errors of the particle filter that implements standard multinomial resampling is $\sqrt{N}$ where N is the number of particles, i.e. the distribution of approximation errors multiplied by $\sqrt{N}$ converges to a

normal distribution as it would if i.i.d. samples were used. A special case of butterfly resampling algorithm parameterized by a natural number $r \geq 2$, which we call radix-r butterfly resampling, exhibits *exotic* scaling behaviour with scaling factor $\sqrt{N/\log_r N}$. This is the cost of keeping the degree of interaction at a fixed level while increasing the number of particles N. Another special case of butterfly resampling, again parameterized by a natural number $r \geq 2$, which we call mixed radix-r butterfly resampling, retains the usual scaling factor $\sqrt{N}$ but with a greater asymptotic variance than that of the standard multinomial resampling. We give the convergence results and central limit theorems for approximation errors in this thesis. Proofs of these results require somewhat more involved techniques than standard machinery used in the analysis of particle filter algorithms and they are given in [51].

1.3. Generic Resampling Algorithm and Interaction Structure

In this section we describe the general form of resampling algorithms and present the concept of interaction structure of a resampling algorithm, throught the thesis we will present the resampling algorithms within this general form. A generic resampling algorithm takes as input a set of input particles $\xi_{\text{in}} = (\xi_{\text{in}}^j)_{j=1}^N$ and their associated weights $w_{\text{in}} = (w_{\text{in}}^j)_{j=1}^N$ and returns a new random set of particles $\xi_{\text{out}} = (\xi_{\text{out}}^j)_{j=1}^{N'}$ such that for each $i \in \{1, \ldots, N'\}$ we have $\xi_{\text{out}}^i = \xi_{\text{in}}^j$ for some $j \in \{1, \ldots, N\}$ and their non-random weights $w_{\text{out}} = (w_{\text{out}}^j)_{j=1}^{N'}$ are chosen such that for any test function φ we have the relation

$$\mathbb{E}\left(\sum_{i=1}^{N'} \frac{w_{\text{out}}^i}{\sum_j w_{\text{out}}^j} \varphi(\xi_{\text{out}}^i)\right) = \sum_{i=1}^{N} \frac{w_{\text{in}}^i}{\sum_j w_{\text{in}}^j} \varphi(\xi_{\text{in}}^i). \tag{1.12}$$

The last equality ensures that, on average, estimates made by resampled particles match the estimates made by input particles, a property which is needed in order to get theoretically justified instances of particle filter algorithms. Note that number of input particles N and output particles N' need not be the same. However we consider particle filtering algorithms where number of particles is kept fixed at every step, so we will assume $N = N'$ throughout the rest of the thesis.

The specification of a resampling algorithm involves choosing a scheme in which the correspondence between input and output particles is given. This correspondence is identified by a random vector of *ancestors* $a = (a^i)_{i=1}^N \in \{1, \dots, N\}^N$ such that $\xi_{\text{out}}^i = \xi_{\text{in}}^{a^i}$. Furthermore, conditional independence structure of this scheme can be depicted by a probabilistic graphical model [53], specifically by a directed bipartite graph where one partite set corresponds to the input particle and other partite set corresponds to the output particles. We call this graph the *interaction structure* of the resampling algorithm. Since conditional distributions of output particles given input particles are discrete distributions with finite supports, they are completely characterized by nonzero probabilities and in the interaction structure of the resampling algorithm, there is an edge from the input particle ξ_{in}^j to the output particle ξ_{out}^i if and only if $\mathbb{P}(a^i = j) \neq 0$.

We will call an interaction structure where the underlying undirected graph is a complete bipartite graph a *full interaction structure*, since all input variables interact with all output variables. Standard multinomial resampling algorithm, of which we investigate the parallel implementation in Section 2.1, exhibits a full interaction structure. On the other hand, we will call an interaction structure where the underlying undirected graph is not a complete bipartite graph a *constrained interaction structure*, see Figure 1.5. A particularly useful class of constrained interaction structures are the ones with underlying graphs having more than one connected component. This kind of constrained interaction structure allows different connected components to be processed in parallel without communication. However, constraining the interaction of resampling mechanism, decreases the quality of the sample as measured by ESS defined in Equation 1.11. We review the idea of resampling under constrained interactions in Chapter 3.

The interaction structure of resampling algorithm is closely related to the communication patterns imposed on the computation units on which the algorithm is implemented. Suppose for example, in a parallel processing system each processor is assigned to resample one output particle. If the resampling algorithm has a full interaction structure, then each processor might need to access any memory location amongst the ones keeping the input particles. In a setting where memory accesses can be pro-

hibitively expensive, as in global memory accesses in GPUs, this kind of interaction structure cannot be efficiently implemented. By contrast, if the resampling algorithm has a constrained interaction structure, then each processor will need to access only a fraction of memory locations. This kind of interaction allows us, for example, to utilize much faster shared memory in GPUs.

To better understand the relation of interaction structure of a resampling algorithm to the communication structure of computational units, we consider a hypothetical situation in which we have a 1-D mesh architecture. 1-D mesh is a distributed computer architecture where we have a one dimensional array of processors such that each processor can communicate directly with only two neighbouring processors [26, Chapter 7]. Suppose that each processor accomodates one particle and resampling of each particle will be carried out in one processor. In this case, an interaction structure as in Figure 1.5. would be implemented easily, because when resampling, a processor will need information only from the neighbouring processors which it can get by direct communication. If there had been an edge from ξ_{in}^1 to ξ_{out}^3 for example, processors 1 and 3 would have to communicate through processor 2, a situation we find undesirable in terms of communication efficiency.

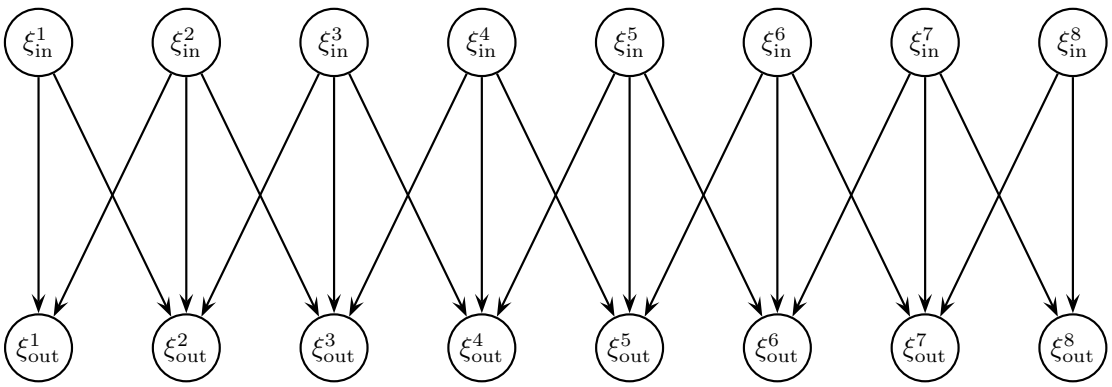


Figure 1.5. An example of a constrained interaction structure.

The main idea of augmented resampling scheme is to enrich the interaction structure of a resampling algorithm by augmenting layers of auxiliary random variables between input and output variables. Although this can remove the conditional independence of output variables given input variables, it can be used to make the flow of information from input variables to output variables more suitable to parallel im-

plementation via layers of constrained interactions while preventing detrimental effects of constrained interactions on ESS. The butterfly resampling algorithm in which the interaction structures between pairs of consecutive layers are characterized by modular congruence relations is proposed within this framework. We present the *augmented resampling* framework in Section 3.1 and *butterfly resampling* algorithm in Section 3.2.

Computational burden of resampling operation can be alleviated by performing it only occasionally, whenever the quality of the particle-weight set, measured by ESS, falls below a prespecified threshold [12]. Authors of [43] generalize this idea by adaptively choosing the interaction structure, hence adjusting the computational cost of resampling, according to ESS. We review the idea of choosing interaction structure adaptively in Chapter 3 along with the idea of resampling under constrained interaction. The same recipe can also be applied within the framework of butterfly resampling and we present the resulting *adaptive butterfly resampling algorithm* in Section 3.3.

1.4. Modern GPU Architectures

In this section we give a brief description of moden GPU architecture and give the details on how this architecture affects our implementation details. We follow the exposition of [34] to identify the limitations and rules-of-thumb to design efficient programs that run on GPUs.

Modern GPU architectures adopt the many-thread paradigm in contrast to multi-core architectures of CPUs. While CPUs are specialized in task parallelism with their multiple processor cores, typically 2-8 cores, that have complex control units, GPUs can accomodate hundreds or thousands of parallel threads that implement simpler instruction sets to provide massive data parallelism (Figure 1.6.). With their massive number of processors, GPUs can yield a tremendous computational horsepower (Figure 1.7.a) and to feed their massively parallel pipelines with poor flow and memory control units they need very high bandwidth for memory transactions (Figure 1.7.b) [54].

Two major platforms for programming modern GPUs are NVIDIA's CUDA plat-

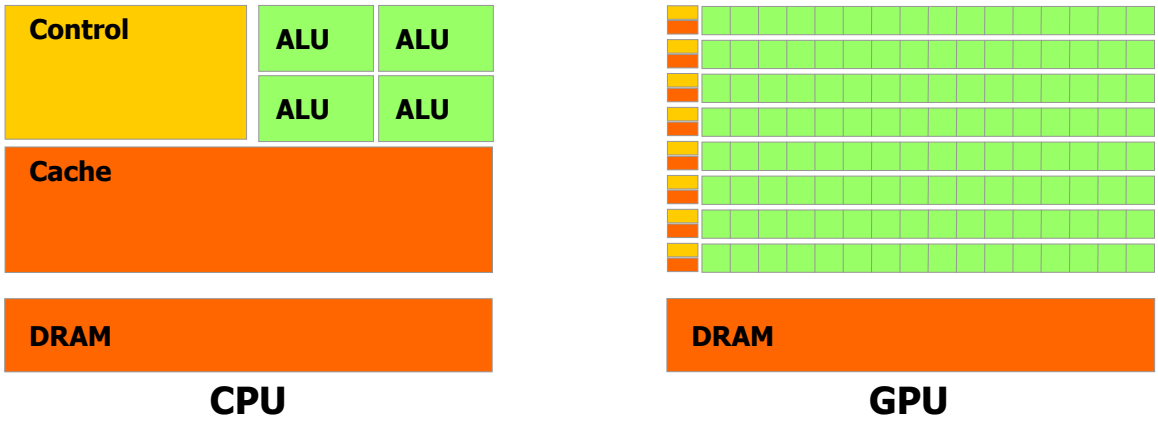
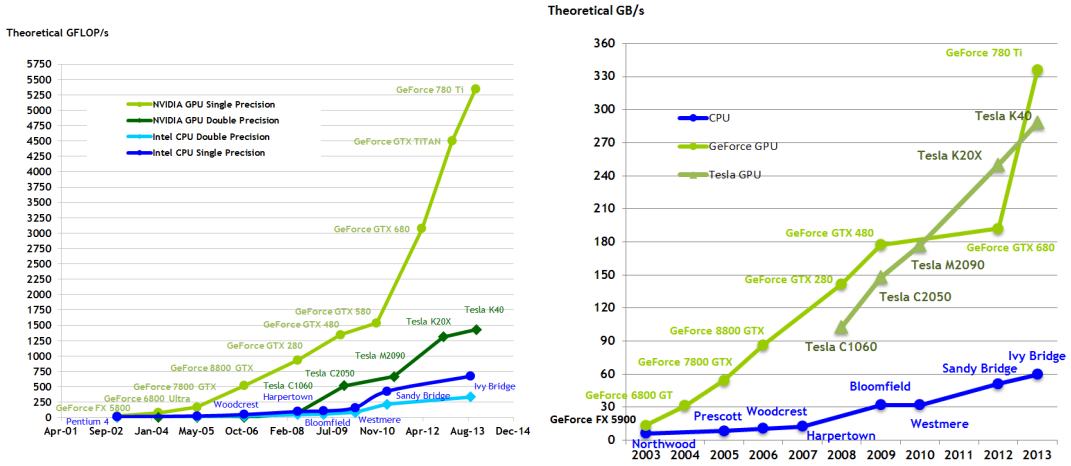


Figure 1.6. CPU architecture vs GPU architecture[1].

form and Khronos Group's OpenCL platform. CUDA programming model is supported by current NVIDIA GPUs. OpenCL, which is jointly developed by major companies in industry including Apple, NVIDIA, AMD etc., targets cross-platform parallel programming and supported by GPUs from different vendors as well as some multicore CPUs or even FPGAs.They both rely on language extensions and runtime APIs [34]. Since, CUDA programming model specifically designed for NVIDIA GPUs and does not target portability as OpenCL, it is easier to learn and write programs in CUDA platform. For this reason, in this thesis, we implement our programs on NVIDIA GPUs using CUDA platform. However, both programming models use very similar language constructs and abstractions [34] and our codes can be adapted to OpenCL platform with little effort.

CUDA programming model implements the single instruction multiple threads (SIMT) model, which is similar to single instruction multiple data (SIMD) model, and has a hierarchical abstract structure on threads provided by thread blocks, barrier synchronizations and shared memory. This hierarchical abstract sructure on threads leads to the partition of problem at hand into coarse subproblems that can be solved in parallel by different thread blocks and then to the partition of these subproblems into finer subproblems that can be solved cooperatively within thread blocks [31]. Each thread block can consist of a maximum of 1024 threads in current specification and

[1]image taken from `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`



(a) Floating point operations per second.

(b) Memory bandwidth.

Figure 1.7. Floating point operations performance and memory bandwidth comparisons of CPUs vs GPUs[2].

thread blocks come together to form the *grid*. Threads and blocks are indexed by built-in variables and they are mapped to hardware components by CUDA runtime system [54]. Threads in the same block can access a shared memory block which is allocated to just one thread block and they can be synchronized by barriers, while threads from different blocks cannot access their respective shared memories and cannot synchronize with each other. Apart from the shared memory, there is also a global memory that can be accessed by all threads, but global memory accesses are much more slower. These constraints impose a local communication structure on threads, which can be represented by a picture as in Figure 1.8. When developing parallel CUDA programs, we should design communication patterns of our algorithms to fit into this local communication structure of CUDA architecture.

A typical CUDA program includes some code that runs on CPU, which is termed *host*, and code that runs on GPU, which is termed *device*. Piece of code that runs on device is called a *kernel* and kernels are launched by host with a thread-thread block-shared memory configuration specified by the programmer. Each kernel launch, in addition to the usual parameters of functions, involves the specification of number of threads per each block, number of thread blocks and optionally the amount of shared

[2]image taken from `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

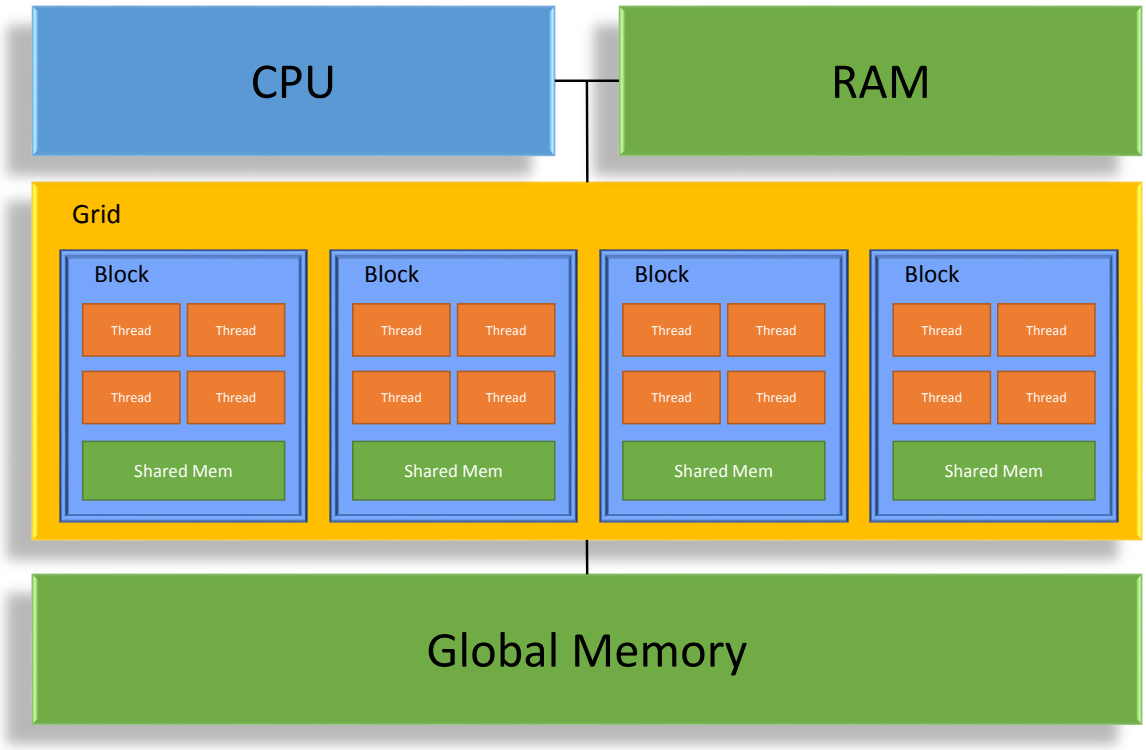


Figure 1.8. Organization of threads and memory in GPU.

memory to be allocated to each thread block. The device has a dedicated memory like the usual RAM, which is called *global* memory and allocation and deallocation of global memory and memory transfers between global memory and host memory or global memories of two devices are performed via special API calls by the host, within a kernel available operations on global memory are limited only to read and write operations.

Other than the local communication structure imposed by thread blocks and shared memory, efficiency of CUDA programs also rely heavily on the structure of control statements, like **if. . . else**, **while** statements etc., in our programs and global memory access patterns of threads. These limitations mainly stem from the physical implementation of SIMT model on the hardware. In CUDA each thread block assigned to a streaming multiprocessor (SM), blocks can not be divided to multiple SMs, but one SM can accomodate more than one thread block. The maximum number of thread blocks that can be accomodated by a SM varies according to GPU models and depends on the number of streaming processors (SP), computation units that share control

logic and instruction cache, each SM have. Each thread block is further divided into *warp*s, basic unit of computation in the hardware level. Within a warp, all threads must execute the same instruction with different data simultaneously. When a branch instruction is performed, threads taking different branches cannot execute the same instruction, this causes to some of the threads in a warp to be idle and hence slow down the execution. This undesirable situation is called *thread divergence* and should be avoided as much as possible by carefully working on thread indices active on a branch.

Another consideration is the global memory access patterns of the program. This is again related to the hardware implementation of the SIMT model. The global memory is implemented on dynamic random access memory (DRAM) which in general has poor latency and to compensate this poor latency very high bandwidth hardware is employed. Efficient utilization of high bandwidth is achieved by fulfilling the memory access requests in bulks, whenever the content of a memory location is requested, a block of consecutive memory locations is accessed. This favors the memory access patterns in which consecutive threads request consecutive memory locations, this type of memory access pattern is called *coalesced access*. In contrast, the memory access patterns in which consecutive threads request random memory locations will hamper the parallelization by poor utilization of the high bandwidth. Since shared memory has better latency compared to global memory, we do not encounter these kind of problems when using shared memory. For this reason, when working on a program in which we need to access memory frequently, we prefer to move the data to shared memory, work on shared memory and move the result back to global memory, whenever this is possible.

To sum up, when developing parallel programs to run on GPUs, we pay attention to above three considerations. First and foremost, in order to fit our algorithm into local structure of GPU architecture imposed by thread block structure, barrier synchronization and shared memory constraint, we should divide our problem into subproblems that can be solved in parallel by thread blocks. These subproblems are further divided into smaller subproblems that can be solved cooperatively by threads

within a thread block. Secondly, for efficient use of massively parallel architecture of GPU, we should refrain from thread divergence by mapping our subproblems carefully to the threads having correct indices. We pay attention to this point especially in computation of cumulative sums, where we use scan algorithm [55]. Lastly, for efficient use of high memory bandwidth of GPU, we should employ coalesced global memory accesses as much as possible. In addition to coalesced global memory accesses, whenever possible we will move the data to shared memory from global memory, process it in the shared memory and copy back to global memory when we are done. We keep in mind that there are numerous factors affecting the performance of GPU programs but we design our algorithms with regard to these three considerations especially.

1.5. Our Contributions

In this thesis, our main contribution is to investigate the GPU implementations of classical resampling algorithms traditionally developed for serial computation and butterfly resampling scheme which is developed to work on a parallel platform with constrained communication structure. A key factor for performance of resampling algorithms in a GPU implementation context is the conditional independence structure of the particles involved, which we formalize as interaction structure of the resampling algorithm. In order for a resampling algorithm to perform well on GPUs, its interaction structure should be mapped to the abstract architecture implied by the communication contraints posed by hardware implementation and programming model of the GPU. Apart from the performance consideration, this interaction structure directly influence the statistical efficiency of the resampling algorithm through variance of asymptotic error fluctuations.

Here, we present classical resampling algorithms in a parallel computation context and discuss their interaction structures. Then we present resampling algorithms with constrained interaction structures and the augmented resampling framework which enables development of resampling algorithms with more parallelizable interaction structures. As an application of augmented resampling framework we present butterfly resampling scheme which can be applied to classical resampling algorithms to generalize

them and focus our attention on its GPU implementation. The key feature of butterfly resampling algorithm is that its sparse interaction structure can be mapped to parallel architectures to facilitate more parallelizable particle filter algorithms. The pay-off of this more parallelizable interaction structure shows up in the asymptotic variance of the error fluctuations. We present central limit theorems for two special cases of butterfly resampling taken from [51], one which exhibits exotic scaling behaviour with a scaling factor of $\sqrt{N/\log N}$ as opposed to $\sqrt{N}$ scaling factor of standard multinomial resampling algorithm and other one retains $\sqrt{N}$ scaling factor of standard multinomial resampling although with a greater asymptotic variance, where N is the number of particles.

We carry out simulations to verify the theoretical results on butterfly resampling, obtain practical guidelines for implementation of butterfly resampling and compare its performance to classical resampling algorithms on GPUs. In these simulation we see that the scaling factor suggested by the theory is verified by results. We use the practical guidelines we obtained from our simulations to efficiently implement our tests for comparisons with classical resampling algorithms. We compare our algorithms first in the context of single step of resampling isolated from a particle filter algorithm, then in the context of two practical implementations of particle filters: bootstrap particle filter and adaptive resampling particle filter. Our results suggest that butterfly resampling scheme can provide speed-up over standard multinomial resampling algorithm, especially in the context of adaptive resampling particle filters. On the other hand, we do not see a noticeable improvement over standard systematic resampling algorithm, which is better in terms of both speed and Monte Carlo error compared to standard multinomial resampling.

We base our observations with asymptotic analysis of time costs of our implementations of resampling algorithms. As a main contribution, we carry out asymptotic time cost analysis for GPU implementations of classical resampling algorithms and their butterfly counterparts for the first time. Our analysis suggests prefix sum computations for all algorithms have $\mathbb{O}(t_S \log_2 N + t_K \log_R N)$ time and sampling part of that standard multinomial resampling has $\mathcal{O}(t_G \log_2 N + t_K)$ time cost in comparison to but-

terfly multinomial resampling algorithm's $\mathcal{O}(t_S \log_2 N + t_K \log_R N)$ time cost, where t_G, t_K, t_S and R are constant terms depending on the hardware specifications of the GPU at hand. Typically we have t_G much greater than t_S and R is much greater than 2, so performance improvement of standard multinomial resampling can be achieved in some cases, since $\log_R N$ term grows much slower than $\log_2 N$ term. On the other hand, sampling part of standard systematic resampling has $\mathcal{O}(ct_G + t_K)$ time cost, where c is a constant term depending on the model, while sampling part of butterfly systematic resampling has $\mathcal{O}(c't_S + t_K \log_R N)$, where c' is another constant depending on the model and parameterization of the resampling algorithm. So, the computation time cost of butterfly systematic resampling is dominated by $\log_R N$ term and this accounts for the speed advantage of standard systematic resampling algorithm over butterfly systematic resampling algorithm.

1.6. Organization of the Thesis

In Chapter 2, we review the classical resampling algorithms: multinomial resampling in Section 2.1, stratified resampling in Section 2.2 and systematic resampling in Section 2.3. We consider these algorithms in a parallel computation context, more specifically we consider their GPU implementations and present their implementations on GPUs. We investigate their interaction structures and discuss their suitability to parallel implementation basing our claims on their interaction structures and communication structure of GPU architectures. We provide asymptotic analysis of time cost for each resampling algorithm. We also give a result on convergence and asymptotic error fluctations for the particle filter that employs multinomial resampling algorithm. This results serves as a base point to which we compare butterfly resampling algorithms. In Section 2.4, we describe two recently proposed resampling algorithms specifically designed for GPU implementation: Metropolis resampling and rejection resampling algorithms and discuss their advantages and disadvantages.

In Chapter 3, first we review the idea of resampling under constrained interactions and present the resampling scheme of αSMC method given in [43]. In Section 3.1, we present the augmented resampling framework recently proposed in [51] in which in-

teraction structure of the resampling is parameterized by a family of matrices and its background theory. We give unbiasedness property and moment bounds for the instance of augmented resampling framework that considers factorizations of the matrix $\mathbf{1}_N$ where $\mathbf{1}_N^{ij} = 1/N$ for all $i, j = 1, \ldots, N$. As an instance of augmented resampling framework, in Section 3.2, we present butterfly resampling algorithm which uses modular congruence relations characterized by a certain form of matrix families to construct sparse interaction structures more prone to parallel implementation. We give GPU implementation of butterfly resampling algorithm which generalize classical resampling algorithms, discuss its interaction structure and provide asymptotic analysis of its time cost. For two special cases of butterfly resampling algorithm, fixed radix-r and mixed radix butterfly resampling algorithms, we give the results on convergence and variance of the asymptotic error fluctuations of the particle filters that employ these algorithms. In Section 3.3, we propose a practical extension to butterfly resampling algorithm based on the idea of adaptive resampling mechanism.

In Chapter 4, we present and discuss our experimental results. In Section 4.1, first we describe the setting in which we evaluate the performance of resampling algorithms. In Section 4.1.2, we verify the theoretical results on the variance of butterfly resampling algorithms by simulation. In Section 4.1.3, we run simulations to obtain practical guidelines for the parameterization of butterfly resampling algorithm. We use these practical guidelines in our implementations of butterfly resampling algorithms in the rest of the experiments. In Section 4.1.4, we compare the performance of GPU implementations of single steps of butterfly resampling algorithm and standard multinomial resampling algorithm in isolation from a particle filter context. In Section 4.2, we compare the performance of GPU implementations of butterfly resampling and multinomial resampling algorithms within two different particle filtering scenarios: in Section 4.2.2 we compare resampling algorithms in bootstrap particle filter scenario where resampling is performed at each stage until the weights become uniform and in Section 4.2.3 in adaptive resampling particle filter scenario where resampling is performed until a certain ESS threshold is exceeded. Finally in Section 4.3, we compare performance of resampling algorithms in a practical context. We perform SMCEM algorithm described in [56] to estimate parameters of a stochastic volatility model with

particle filter implementations employing butterfly resampling algorithm and multinomial resampling algorithm.

In Chapter 5 we present our conclusions and possible directions for future work. In Appendix A, we present the implementation details of performance critical parts of the algorithms. In Section A.1, we give the implementation details of scan algorithm which is used for prefix sum computation and essential for all of the resampling algorithms we discussed and present the code for its GPU implementation. In Section A.2, we discuss the efficient implementation of butterfly resampling algorithm on GPUs. We present strategies for reducing the cost of prefix sum computation, using simple bit manipulations for fast computation of mapping between threads and particles and using tiling method for coalescing the global memory accesses and give GPU code for implementations of these strategies. Finally in Section A.3, we discuss the issue of movement of particles in global memory after resampling. We focus on two strategies: double buffer strategy and in-place propagation of particle and present GPU code of their implementations.

2. PARALLELIZATION OF CLASSICAL RESAMPLING ALGORITHMS

The first instance of bootstrap particle filter [10] implements the multinomial resampling algorithm, which is basically sampling with replacement with probabilities proportional to the weights. However, there are other approaches some of which are more superior to the multinomial resampling in terms of Monte Carlo variance and computation cost. In this thesis, we consider two other classical resampling algorithms: stratified resampling and systematic resampling. We present all algorithms in a parallel computation context, more specifically, we focus on their implementations on GPUs. We also give a simple analysis of their computation time on GPUs, in our analysis we assume that we have N threads that can run in parallel when we have N particles. In the following three sections, we give the descriptions of classical resampling algorithms, then we give the results of our simple computational cost analysis and remark on their interaction structures. We also give a theorem without proof on the convergence properties of particle filter that use multinomial resampling algorithm.

For a survey of classical resampling algorithms in a serial computation context see for example [57, 58]. A recent work [42], presents an extensive account on GPU implementations of classical resampling algorithms and proposes two new algorithms that are specifically designed for implementation on GPUs: Metropolis resampling and rejection resampling. We present these algorithms here as well, but our reference point will be the classical resampling algorithms, since the results on classical algorithms are more established. In our presentation of GPU implementations of resampling algorithms, we follow the style adopted in [42].

2.1. Multinomial Resampling

A widely used algorithm for resampling is *multinomial resampling* in which ancestors $a = (a^i)_{i=1}^N$ are independently sampled from a categorical distribution with

support $\{1, \ldots, N\}$ and corresponding probabilities given by the normalized weights $(p^k = w^k / \sum_{i=1}^{N} w^i)_{k=1}^{N}$. In order to satisfy Equation 1.12, output weights of multinomial resampling algorithm are all set to $w_{\text{out}}^i = W^N/N = \frac{1}{N} \sum_{j=1}^{N} w_{\text{in}}^j, i = 1, \ldots, N$. Since each output particle can have any of the input particles as its ancestor, the conditional independence structure for multinomial resampling can be characterized by the complete bipartite directed graph in Figure 2.1. and main challenges in parallelizing multinomial resampling manifest themselves in the dense structure of this graph.

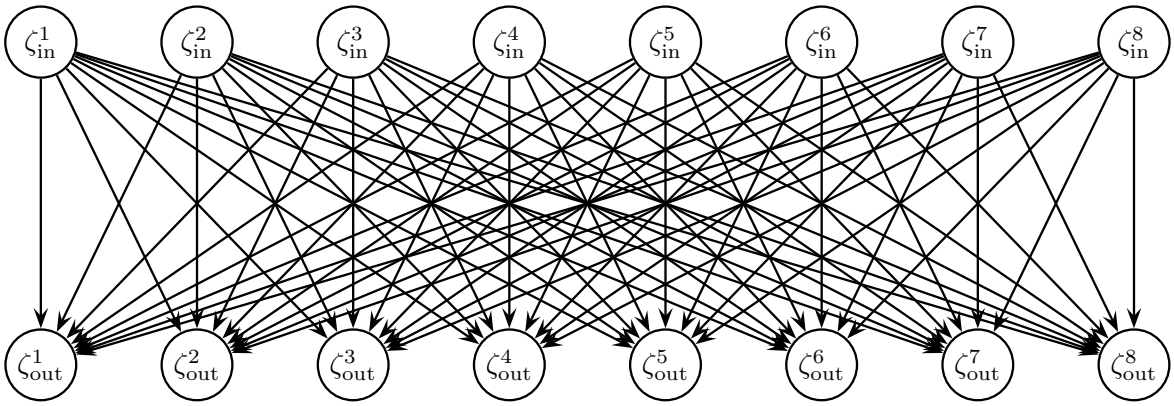


Figure 2.1. Full interaction structure of standard multinomial resampling.

The standard approach for sampling from a categorical distribution with cumulative weights $W = (W^k = \sum_{i=1}^{k} w^i)_{k=1}^{N}$ is inversion sampling [59]. This subroutine will be used in other resampling algorithms too and is given in Figure 2.3. GPU implementation of multinomial resampling involves computation of the cumulative weights vector W and subsequent parallel calls to the inversion sampling subroutine that takes the cumulative weight vector as input. The pseudocode code for this method, called MultinomialResample, is shown in Figure 2.2.

In Figure 2.2., we fully parallelize calls to inversion sampling, since each ancestor is sampled independently. However, there are three key obstacles for scaling up this method for very large N, mainly caused by the complete bipartite interaction structure imposed by multinomial resampling:

(i) We need to compute prefix sum over whole weight array, which necessitates barrier synchronization of all threads. However this is not possible when working on arrays whose number of elements is greater than the maximum number of threads

per block, so we need multiple kernel calls to perform this prefix sum.

(ii) We need to perform N binary search operations over whole cumulative weight array which does not fit to the shared memory in general. The unavoidable random moves of binary search cause both thread divergence and heavy traffic on global memory,

(iii) We need to use a double buffer strategy for the propagation of particles. When updating particles after resampling step, in order to copy particles in place we have to synchronize all threads, which necessitates barrier synchronization of all threads. Since this is not possible on GPUs we have to use different arrays for input and output particles.

1: **function** MULTINOMIALRESAMPLE(w_{in}, x_{in})
2: $W \leftarrow$ PREFIXSUM(w_{in})
3: **for each** $i \in \{1, \ldots, N\}$ **do**
4: $j^i \leftarrow$ INVERSIONSAMPLING($W, \{1, \ldots, N\}$) //Sample ancestors
5: $x^i_{\text{out}} = x^{j^i}_{\text{in}}$ and $w^i_{\text{out}} \leftarrow \dfrac{\sum_{j=1}^{N} w^j_{\text{in}}}{N}$ //Update particles and weights
6: **end for**
7: **return** $w_{\text{out}}, x_{\text{out}}$
8: **end function**

Figure 2.2. GPU Implementation of Multinomial Resampling.

1: **function** INVERSIONSAMPLING($W = \{W^1, \ldots, W^N\}, S = \{S_1, \ldots, S_N\}$)
2: $u \sim \mathcal{U}[0, W^N)$ //Sample uniformly on $[0, W^N)$
3: $j \leftarrow$ LOWERBOUND(u, W) //Locate the sampled index in $[N]$
4: **return** S_j //Return the sampled state
5: **end function**
6: LOWERBOUND(u, W) **returns** $\text{argmin}_{i \in [N]}\{u < W^i\}$ //Binary search within W

Figure 2.3. Inversion Sampling.

The GPU implementation of Figure 2.2. involves several kernel calls: one kernel call for the sampling part, $\lceil \log_{r_{\max}} N \rceil$ kernel calls for prefix sum, where $r_{\max}$ is the maximum number of threads per block, and one for update part. A simple analysis

of Figure 2.2. shows that it takes $\mathcal{O}(\log N)$ time to perform multinomial resampling on GPU. However, to compare it with butterfly resampling algorithms we carry out a more detailed analysis. Let t_G be the average time cost of random global memory accesses, t_S be the average time cost of random shared memory access, we make this distinction since global memory accesses are substantially more expensive than shared memory accesses, and let t_K be the average time cost of kernel launch of the sampling part. Then it takes $\mathcal{O}(t_G \log_2 N + t_K)$ time to perform sampling part of the multinomial resampling on GPU, where $t_G \log_2 N$ term comes from the binary search carried out in LowerBound function. Here, we take the terms involving kernel call times, and global memory access times since butterfly resampling algorithms differ from standard multinomial resampling algorithm in these aspects. On the other hand, PrefixSum function takes $\mathcal{O}(t_S \log_2 N + t_K \log_{r_{\max}} N)$ time, where $t_S \log_2 N$ term comes from the scan algorithm [55] which is a standard procedure for computing prefix sums on parallel architectures. We present the implementation details of scan algorithm in Appendix A.1.

Straightforward implementation of multinomial resampling algorithm most naturally delivers an ancestors vector $a = (a^1, \ldots, a^N)$. Then the particles can be updated by the formula $x^i_{\text{out}} = x^{a^i}_{\text{in}}$. In order to perform in place propagation of particles, first we need to copy ancestor input particle into a local buffer, synchronize all threads then copy the content of local buffer into memory location for particle associated to the thread. Synchronization is imperative in order to avoid a race condition in a parallel architecture: a memory location that will be read by a thread should not be written by another thread before the read operation by the former thread. However, barrier synchronization of all threads is not possible in CUDA architecture. To avoid race condition we have to either ensure that ancestor vector will not lead to a race condition or use a double buffer strategy. If the ancestor vector satisfies the condition that for all $i = 1, \ldots, N$ if $a^j = i$ for some j we should also have $a^i = i$, then this ancestor vector permits the particles to be propagated without race condition, hence does not necessitate synchronization. Since multinomial resampling is not guaranteed to deliver such an ancestor vector, the resulting ancestor vector should be permuted to satisfy this condition. This permutation can be performed in parallel [42], yet it charges an

additional computational cost. These considerations on propagation of particles, apply in the same way to stratified and systematic resampling algorithms as well. In our particle filter implementations with butterfly resampling, we use a double buffer strategy in which one array keeps the input particles and other one keeps the output particles. We give the implementation details regarding the propagation of particles in Appendix A.3.

Although full interaction structure of multinomial resampling complicates the parallelization of resampling part, its independence structure, namely the property that $\mathbb{P}(a^{i_1} = j_1, a^{i_2} = j_2) = \mathbb{P}(a^{i_1} = j_1)\mathbb{P}(a^{i_2} = j_2)$, leads to the $\sqrt{N}$ scaling of asymptotic error fluctuations of estimators obtained from particle filter. This is the same scaling behaviour as we would get from i.i.d. samples from the distribution of interest. Considering the empirical measures π_n^N and $\hat{\pi}_n^N$ defined in equations 1.5,1.7 and associated estimators defined in equations 1.8,1.9, we have the following theorem given in [51]:

Theorem 2.1. *For any $n \geq 0$ and bounded measurable function $\varphi : \mathsf{X} \to \mathbb{R}$, define*

$$\begin{aligned}
\sigma_0^2(\varphi) &:= \pi_0((\varphi - \pi_0(\varphi))^2), \\
\sigma_n^2(\varphi) &:= \hat{\sigma}_{n-1}^2(f(\varphi)) + \hat{\pi}_{n-1}(f((\varphi - f(\varphi))^2)),\ n \geq 1, \\
\hat{\sigma}_n(\varphi) &:= \hat{\pi}_n((\varphi - \hat{\pi}_n(\varphi))^2) + \pi_n(g_n)^{-2}\sigma_n^2(g_n(\varphi - \hat{\pi}(\varphi))),\ n \geq 0,
\end{aligned}$$

where $f(\varphi)(x) = \int \varphi(x')f(x, dx')$ and assume these quantities are strictly positive. Then, for output empirical measures of particle filter with multinomial resampling we have as $N \to \infty$

$$\pi_n^N(\varphi) - \pi_n(\varphi) \xrightarrow{a.s.} 0,\ \sqrt{N}(\pi_n^N(\varphi) - \pi_n(\varphi)) \xrightarrow{dist} \mathcal{N}(0, \sigma_n^2(\varphi)), \tag{2.1}$$

$$\hat{\pi}_n^N(\varphi) - \hat{\pi}_n(\varphi) \xrightarrow{a.s.} 0,\ \sqrt{N}(\hat{\pi}_n^N(\varphi) - \hat{\pi}_n(\varphi)) \xrightarrow{dist} \mathcal{N}(0, \hat{\sigma}_n^2(\varphi)). \tag{2.2}$$

This theorem will serve as a reference to which we will compare the convergence results of butterfly resampling algorithms. It can be proved by direct applications of

results of e.g. [19] to the empirical distributions that we are interested in.

2.2. Stratified Resampling

Stratified resampling is first proposed in [60] and is used to reduce the Monte Carlo variance associated to the multinomial resampling. It is based on the ideas used in survey sampling and consists in pre-partitioning the interval $[0, W^N)$ into N disjoint intervals with equal lengths $[0, W^N/N), [W^N/N, 2W^N/N), \ldots, [(N-1)W^N/N, W^N)$ [57]. Then we sample N numbers, u^i from each of the interval $[(i-1)W^N/N, iW^N/N)$ for all $i = 1, \ldots, N$ uniformly and independently. Finally, we use these random numbers to generate our ancestors, such that

$$a^i = j \text{ if } u^i \in [W^{j-1}, W^j).$$

Note that, according to this definition, permutations of the input particles and weights will produce a different distribution. When $[kW^N/N, (k+1)W^N/N) \subset [W^{j-1}, W^j)$ for some $j, k \in \{1, \ldots, N\}$, then with probability 1 we will have $a^k = j$. This property makes assures that particles with large weights will be resampled and decrease the variance of the resampling process defined as:

$$\text{Var}\left(\frac{1}{N}\sum_{i=1}^{N}\varphi(x_{\text{out}}^i)\middle| x_{\text{in}}\right) = \mathbb{E}\left(\left(\frac{1}{N}\sum_{i=1}^{N}\varphi(x_{\text{out}}^i) - \sum_{i=1}^{N}\frac{w_{\text{in}}^i}{\sum_j w_{\text{in}}^j}\varphi(x_{\text{in}}^i)\right)^2\middle| x_{\text{in}}\right). \quad (2.3)$$

It can be shown that the variance of the stratified resampling is always less than the variance of multinomial resampling [57].

Stratified resampling algorithm more naturally delivers the cumulative offsprings vector $O = (O^i \coloneqq \sum_{j=1}^{N} \mathbb{I}(a^j \leq i))_{i=1}^N$ where $\mathbb{I}(A)$ denotes the indicator function of the event A and does not necessitate a binary search within cumulative weights vector W [42]. The cumulative offspring O^i keeps the total number of offspring of input particles with index less than or equal to i. This simplifies the parallelization procedure of stratified resampling algorithm by removing the cost associated to the binary search.

However, it necessitates an extra step for the conversion of cumulative offspring vector to ancestors vector. Pseudocode for the GPU implementation of stratified resampling is given in Figure 2.4..

1: **function** StratifiedResample(w_{in}, x_{in})
2: $W \leftarrow$ PrefixSum(w_{in})
3: **for each** $i \in \{1, \dots, N\}$ **do**
4: $r^i \leftarrow \dfrac{NW^i}{W^N}$
5: $k^i \leftarrow \min(N, \lfloor r^i \rfloor + 1)$
6: $u^i \sim \mathcal{U}[0, 1)$
7: $O^i \leftarrow \min(N, \lfloor r^i + u^{k^i} \rfloor)$ and $w^i_{\text{out}} \leftarrow \dfrac{\sum_{j=1}^N w^j_{\text{in}}}{N}$
8: **end for**
9: $j \leftarrow$ CumulativeOffspringToAncestors(O)
10: **for each** $i \in \{1, \dots, N\}$ **do**
11: $x^i_{\text{out}} \leftarrow x^{j^i}_{\text{in}}$
12: **end for**
13: **return** $w_{\text{out}}, x_{\text{out}}$
14: **end function**

Figure 2.4. GPU Implementation of Stratified Resampling.

As in case of multinomial resampling, GPU implementation of Figure 2.4. involves several kernel calls: one kernel call for the sampling part, $\lceil \log_{r_{\max}} N \rceil$ kernel calls for prefix sum, one for conversion of cumulative offsprings to ancestors and one for update part. As we pointed out above, sampling kernel does not involve calls to a binary search procedure. This saves us from a time cost of $\mathcal{O}(t_G \log N)$, so the sampling of cumulative offspring takes constant time $\mathcal{O}(t_G + t_K)$. In Figure 2.5., we give the pseudocode of the GPU implementation of conversion of cumulative offsprings to ancestors. The main bottleneck of the computation in Figure 2.5. is the **for** loop in line 9 which takes on average $N \frac{\max_i \{w^i_{\text{in}}\}}{W^N} t_G$ time on GPU for a fixed weight vector w_{in} and particle number N. In particle filter implementations we have $w^i_{\text{in}} = g(\xi^i_{\text{in}})$ where g is a bounded function and we have $\frac{W^N}{N} \to Z < \infty$ for some constant Z. So, after the computation of cumulative weights vector, it takes $\mathcal{O}(ct_G + t_K)$ time to produce the ancestors vector

```
1: function CumulativeOffspringToAncestors(O)
2:     for each i ∈ {1, ..., N} do
3:         if i = 1 then
4:             s ← 0
5:         else
6:             s ← O^{i−1}
7:         end if
8:         o^i ← O^i − s                    //Number of offsprings of ith particle
9:         for j = 1, ..., o^i do
10:            a^{s+j} ← i                  //Set ancestors to the ith particle
11:        end for
12:    end for
13:    return a
14: end function
```

Figure 2.5. Conversion of Cumulative Offsprings to Ancestors.

in GPU implementation of stratified resampling algorithm with $c = \sup_x g(x)/Z$.

The mechanism of stratified resampling impose a full interaction structure on the particles like multinomial resampling, it does not put restrictions on the ancestoral relations of the particles and we need to compute the whole cumulative weights vector. However, conditioned on the input weights w_{in}, we no longer have a full interaction structure, depending on the ordering of the input particles and weights, each output particle is sampled from a small set of input particles. So, the foremost obstacle in parallelization of the stratified resampling algorithm is the computation of cumulative weight array. Another point that would be detrimental to performance is the **for** loop in line 9 of Figure 2.4., which causes thread divergence and in case of very unbalanced weight practically serializes the algorithm.

2.3. Systematic Resampling

Systematic resampling algorithm, first introduced in [14], further reduces the randomness by using only one random number and like stratified resampling it is, in general, used to reduce the Monte Carlo variance associated to the resampling process. It is the same the as the stratified resampling, except the way u^is are generated: in systematic resampling we sample a random number $u \sim \mathcal{U}[0, W^N/N)$ and set $u^i = (i-1)W^N/N + u$. Then ancestors are generated again by using the following rule:

$$a^i = j \text{ if } u^i \in [W^{j-1}, W^j).$$

Since u^is are dependent, it is much more complicated to provide an expression for variance of systematic resampling as defined in Equation 2.3 and although according to empirical results, in general, it is less than variance of multinomial resampling, it is possible to construct examples in which variance of systematic resampling is not only greater than that of multinomial resampling but does not even decrease with the number of particles, contrary to the other resampling schemes [57]. Despite this, simulations suggest that variance of systematic resampling is less than all other classical algorithms in most practical situations.

We give the pseudocode for systematic resampling that takes an additional input of random offset u in Figure 2.6. Note that its only difference from Figure 2.4. is the generation of u^is performed in line 6. So, it takes $\mathcal{O}(t_G + t_K)$ time to produce the offspring vector, but since addition operation takes less time than random sampling, the cost associated to line 6 of Figure 2.6., which shows up in the constant factor, will be slightly less compared to stratified resampling. Other points in the analysis of stratified resampling algorithm applies to the systematic resampling in the same way and we have a total of $\mathcal{O}(ct_G + t_K)$ time cost associated to sampling cumulative offsprings and producing ancestors from this cumulative offsprings vector. Along with computational cost analysis, remarks on interaction structure of stratified resampling algorithm applies to systematic resampling algorithm as well.

function SystematicResample(w_{in}, x_{in}, u)
  $W \leftarrow$ PrefixSum(w_{in})
  for each $i \in \{1, \ldots, N\}$ **do**
    $r^i \leftarrow \dfrac{NW^i}{W^N}$
    $k^i \leftarrow \min(N, \lfloor r^i \rfloor + 1)$
    $u^i \leftarrow u$
    $O^i \leftarrow \min(N, \lfloor r^i + u^{k^i} \rfloor)$ and $w^i_{\text{out}} \leftarrow \dfrac{\sum_{j=1}^{N} w^j_{\text{in}}}{N}$
  end for
  $j \leftarrow$ CumulativeOffspringToAncestors(O)
  for each $i \in \{1, \ldots, N\}$ **do**
    $x^i_{\text{out}} \leftarrow x^{j^i}_{\text{in}}$
  end for
  return $w_{\text{out}}, x_{\text{out}}$
end function

Figure 2.6. GPU Implementation of Systematic Resampling.

2.4. Metropolis and Rejection Resampling

In [42] Murray et. al. investigate GPU implementations of classical resampling algorithms we reviewed in previous sections and propose two new algorithms that are claimed to better suit to GPU architecture. Metropolis resampling scheme runs an independent Markov chain of which the stationary distribution is categorical distribution on $\{1, \ldots, N\}$ with probabilities proportional to the input weights, i.e. the distribution $\sum_{i=1}^{N} \frac{w^i_{\text{in}}}{\sum_j w^j_{\text{in}}} \delta_i$, for each particle and stops all chains after a presribed number of steps B given as a parameter to the algorithm. The guidelines for choosing the number B is given in the original paper and involves computation of $\mathbb{E}(\bar{w})/w_{\max}$ where $\bar{w}$ is the average weight and $w_{\max}$ is the maximum value that the weight can attain. Closed form expressions for these terms are not available in general and they can be estimated from the input weights, hence a preliminary step for estimation of these terms might be needed.

The pseudocode for Metropolis resampling is given in Figure 2.7. Note that Figure 2.7. does not involve a prefix sum computation, this reduces the total computation cost substantially. If we do not need to compute the observation likelihood, Z_n, weight update scheme also can be changed so that the sum of weights need not be computed. In this case output weights can be set to some constant value like 1 or $1/N$. However, GPU implementation of Metropolis resampling can suffer from completely random global memory accesses needed in order to compute acceptance ratio at line 7. Another drawback is that the distribution of resulting ancestors is not the target distribution $\sum_{i=1}^{N} \frac{w_{\text{in}}^i}{\sum_j w_{\text{in}}^j} \delta_i$, since Markov chains have to be stopped at a finite step. This might complicate the functionality of particle filter, as the theory assumes that the resampling operation satisfies the unbiasedness property characterized by Equation 1.12.

```
1:  function MetropolisResample(w_in, x_in, B)
2:      for each i ∈ {1, ..., N} do
3:          j^i ← i
4:          for n = 1, ..., B do
5:              u ~ U([0, 1))
6:              k ~ U({1, ..., N})
7:              if u ≤ w_in^k / w_in^{j^i} then
8:                  j^i ← k
9:              end if
10:         end for
11:         x_out^i ← x_in^{j^i} and w_out^i ← (Σ_{j=1}^N w_in^j) / N
12:     end for
13:     return w_out, x_out
14: end function
```

Figure 2.7. GPU Implementation of Metropolis Resampling.

The second alternative to the classical resampling algorithms is to use rejection method [59] to generate the ancestors. Rejection method is based on the idea of generating uniform samples from the area under the density of the random variable of interest and then projecting this point to the state space. In order to generate uniform

samples from the area under density function, first, we generate uniform samples from a region that contains this area and easy to sample from, then each sample is either kept, accepted, if it falls into the area under density function or thrown away, rejected, otherwise.

Pseudocode for rejection resampling is given in Figure 2.8. Like Metropolis resampling, rejection resampling does not necessitate the computation of prefix sum of weight but needs the value $w_{\max}$ as a parameter. Again, this value can be estimated from the input weights by a preliminary step if it is not available from the closed form expressions. On the other hand, rejection resampling does not exhibit the problematic bias property of Metropolis resampling, samples generated by this procedures are exact samples from the target distribution $\sum_{i=1}^{N} \frac{w_{\text{in}}^i}{\sum_j w_{\text{in}}^j} \delta_i$. While avoiding prefix sum computation eliminates its heavy computational burden, variable length **while** loop in line 5 of Figure 2.8. causes severe thread divergence and random global memory accesses needed to compute the acceptance ratio leads to heavy memory traffic.

1: **function** REJECTIONRESAMPLE(w_{in}, $x_{\text{in}}, w_{\max}$)
2: **for each** $i \in \{1, \ldots, N\}$ **do**
3: $j^i \leftarrow i$
4: $u \sim \mathcal{U}([0, 1))$
5: **while** $u > w_{\text{in}}^{j^i} / w_{\max}$ **do**
6: $j^i \sim \mathcal{U}(\{1, \ldots, N\})$
7: $u \sim \mathcal{U}([0, 1))$
8: **end while**
9: $x_{\text{out}}^i \leftarrow x_{\text{in}}^{j^i}$ and $w_{\text{out}}^i \leftarrow \dfrac{\sum_{j=1}^{N} w_{\text{in}}^j}{N}$
10: **end for**
11: **return** $w_{\text{out}}, x_{\text{out}}$
12: **end function**

Figure 2.8. GPU Implementation of Rejection Resampling.

3. PARALLEL RESAMPLING UNDER CONSTRAINED INTERACTION

In their recent work [43] Whiteley *et al.* showed that particle filters still *work* under constrained interactions amongst particles when resampling, provided that these constraints satisfy certain conditions and propose αSMC algorithm that makes use of this property. This algorithm uses a family of stochastic matrices, $\mathbb{A}_N = \{\alpha_k\}_{k\geq 0}$, called α matrices to parameterize interaction structure of the resampling algorithm.

For a given particle-weight set $(\xi_{\text{in}}, w_{\text{in}})$ of size N, resampling scheme of αSMC algorithm first specifies the interaction structure by a stochastic matrix α chosen from the set $\mathbb{A}_N$ according to some deterministic function of $(\xi_{\text{in}}, w_{\text{in}})$, and then each of the ancestors $a = \{a^i\}_{i=1}^N$ are sampled independently from the family of categorical distributions for which the probabilities are given as $\mathbb{P}(a^i = j) = \alpha^{ij} w_{\text{in}}^j / \sum_{k=1} \alpha^{ik} w_{\text{in}}^k$. Output weights of this resampling scheme are set by the equation $w_{\text{out}}^i = \sum_{j=1}^N \alpha^{ij} w_{\text{in}}^j$, which satisfies the unbiasedness property given in Equation 1.12.

Three well-known particle filter algorithms, Sequential Importance Sampling (SIS), BPF and adaptive resampling particle filter (ARPF) algorithms are special cases of αSMC where corresponding families of α matrices are $\mathbb{A}_N = \{I_N\}$, $\mathbb{A}_N = \{\mathbf{1}_N\}$, $\mathbb{A}_N = \{I_N, \mathbf{1}_N\}$ respectively.

To see the influence of α matrix on the interaction structure of resampling, note that an α matrix fully characterize the conditional independence structure of input and output particles : a positive element in the position (i, j) of the α matrix indicates interaction between ith output particle and jth input particle, while a zero in the position (i, j) of the α matrix indicates no interaction between ith output particle and jth input particle. More formally, if $\alpha^{ij} = 0$, then we have $\mathbb{P}(\xi_{\text{out}}^i | \xi_{\text{in}}) = \mathbb{P}(\xi_{\text{out}}^i | \xi_{\text{in}}^{1:N\setminus j})$. This observation points that using sparse α matrices leads to resampling mechanisms with more parallelizable sparse interaction structures.

Whiteley et al. show that the control of ESS, which naturally arises in the theoretical analysis of the particle filters, through resampling under constrained interactions gives rise to a class of particle filter algorithms that can be efficiently implemented on parallel platforms without compromising its key statistical properties, namely the convergence and stability.

General form of the resampling part of the αSMC algorithm that controls the ESS is given in Figure 3.1. Note that, we do not specify the details of how to choose α matrix from the set $\mathbb{A}_N$. A set of different strategies for this are given in the original paper. Although line 4 of Figure 3.1. suggests that a prefix sum is computed for each particle, using special forms of α matrices a data reuse approach is employed. For example, if i_1th and i_2th rows of α matrix are the same, then associated cumulative sum need not be computed twice.

1: **function** αSMCRESAMPLE(w_{in}, x_{in}, $\mathbb{A}_N$, $\tau \in [1, N]$)
2: Choose $\alpha_k \in \mathbb{A}_N$ s.t. $\mathcal{E}^N(\alpha_k w_{\text{in}}) \geq \tau$ //Choose interaction structure ensuring ESS bound
3: **for each** $i \in \{1, \ldots, N\}$ **do**
4: $W_i \leftarrow \text{PREFIXSUM}(\alpha_k^{(i,:)} \odot w_{\text{in}}^T)$ //Compute sampling distribution for ith particle
5: $j^i \leftarrow \text{INVERSIONSAMPLING}(W_i, [N])$ //Sample ancestors independently
6: $x_{\text{out}}^i = x_{\text{in}}^{j^i}$ and $w_{\text{out}}^i \leftarrow \sum_{j=1}^N \alpha_k^{ij} w_{\text{in}}^j$ //Update particles and weights
7: **end for**
8: **return** $w_{\text{out}}, x_{\text{out}}$
9: **end function**

Figure 3.1. GPU Implementation of αSMC Resampling.

An example strategy for choosing α matrices, when $N = 2^m$ for some $m \in \mathbb{N}$, is setting $\mathbb{A}_N = \{\alpha_k = I_{2^{m-k}} \otimes \mathbf{1}_{2^k} : 0 \leq k \leq m\}$, the set of block diagonal matrices with 2^{m-k} blocks where each block is in the form of $\mathbf{1}_{2^k}$, and choosing α_k with smallest k such that the ESS after resampling using this matrix is above a given threshold. Note that $\alpha_0 = I_N$ and $\alpha_m = \mathbf{1}_N$, so this is a generalization of adaptive resampling

strategy of ARPF. This strategy corresponds to pairing up particles to form 2^{m-1} groups, pairing up those groups to form 2^{m-2} groups and so on, until resampling under these groupings produce an ESS value above the given threshold. Here, block diagonal structure of α matrices removes the interaction between different blocks of particles and allows different blocks of particles to be resampled in parallel. An example of this kind of constrained interaction structure is depicted in Figure 3.2. in which we have 2 blocks of 4 particles each.

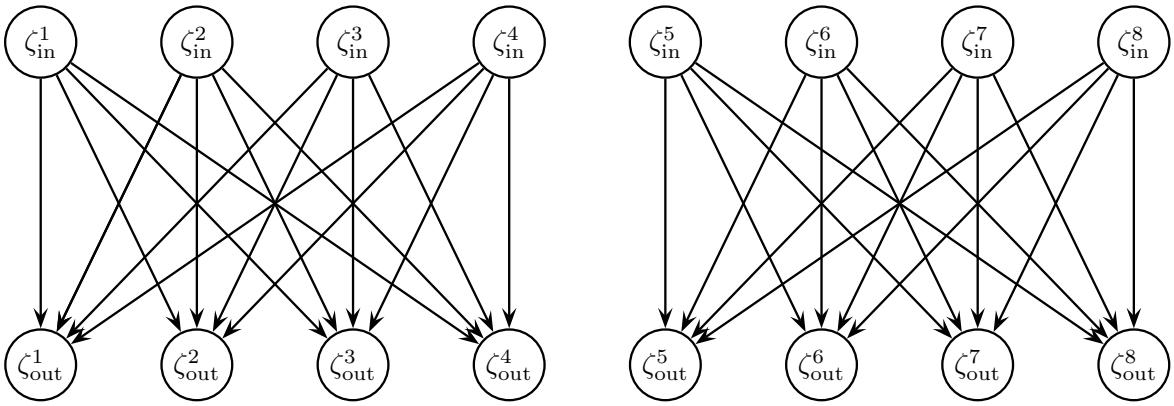


Figure 3.2. Constrained interaction structure of αSMC resampling.

Shortcomings of multinomial resampling applies also to αSMC resampling. Although we need full interaction occasionally, we may still need to implement it on a platform that allows only limited interactions. This force us to come up with resampling algorithms that can keep the degree of interaction at a certain level regardless of the number of particles we need. In the next section we present a recently proposed general framework that enables us to tailor the interaction structure of resampling algorithms to comply with the limitations imposed by parallel platforms.

3.1. Augmented Resampling

Constrained interaction framework of αSMC is further generalized by considering certain m-fold factorizations of stochastic matrices. Let α be a stochastic matrix and $(A_1, \ldots, A_m)$ be a sequence of m stochastic matrices such that $\prod_{i=1}^{m} A_i = \alpha$. The approach of the augmented resampling framework is to augment layers of auxiliary random variables $(\xi_k)_{k=0}^{m}$ to the resampling process such that $\xi_0 = \xi_{\text{in}}$, $\xi_m = \xi_{\text{out}}$ and interaction structure between successive layers are controlled by the matrices $(A_k)_{k=1}^{m}$.

Pseudocode for augmented resampling is given in Figure 3.3. Note that, with $m = 1$ and $A_1 = \mathbf{1}_N$, augmented resampling algorithm corresponds to standard multinomial resampling.

function AUGMENTEDRESAMPLE(w_{in}, x_{in}, $(A_i)_{i=1}^m$)
  $w_0 \leftarrow w_{\text{in}}$ and $\xi_0 \leftarrow x_{\text{in}}$
  for $k \in \{1, \ldots, m\}$ **do**
    for each $i \in \{1, \ldots, N\}$ **do**
      $W_i \leftarrow \text{PREFIXSUM}(A_k^{(i,:)} \odot w_{k-1}^T)$ //Compute sampling distribution
      $j^i \leftarrow \text{INVERSIONSAMPLING}(W_i, [N])$ //Sample ancestors independently
      $\xi_k^i = \xi_{k-1}^{j^i}$ and $w_k^i \leftarrow \sum_{j=1}^N A_k^{ij} w_{k-1}^j$ //Update particles and weights
    end for
  end for
  $w_{\text{out}} \leftarrow w_m, x_{\text{out}} \leftarrow \xi_m$
  return $w_{\text{out}}, x_{\text{out}}$
end function

Figure 3.3. GPU Implementation of Augmented Resampling.

A salient feature of augmented resampling is that it can preserve the marginal distribution of output particles given input particles and maintain the quality of particle-weight set as measured by ESS while making the resampling mechanism more amenable to parallel implementation via sparse factorizations of α matrix. More formally, for a given α such that $\alpha = \prod_{k=1}^m A_m$ where $(A_k)_{k=1}^m$ is a sequence of stochastic matrices and an input particle-weight set $(w_{\text{in}}, x_{\text{in}})$, for the output particle-weight set $(w_{\text{out}}, x_{\text{out}})$ of Figure 3.3., we have

$$\mathbb{P}(x_{\text{out}}^i = x_{\text{in}}^j) = \alpha^{ij} w^j / \sum_k \alpha^{ik} w^k \text{ and } w_{\text{out}}^i = \sum_k \alpha^{ik} w_{\text{in}}^k. \tag{3.1}$$

The first part of Equation 3.1 can be used to obtain lack-of-bias property for the case $\alpha = \mathbf{1}_N$ and second part makes sure that ESS bound maintained by using α matrix in αSMC resampling is also satisfied in augmented resampling. Equation 3.1 can be

obtained by repeated applications of following proposition:

Proposition 3.1. *Let w_0 be a weight set, A, B stochastic matrices, $a_1 \in \{1,\dots,N\}^N$ such that*

$$a_1^i \sim \sum_{j=1}^{N} \frac{A^{ij} w_0^j}{\sum_l A^{il} w_0^l} \delta_j(\cdot) \text{ and } w_1 = A w_0^T,$$

$a_2 \in \{1,\dots,N\}^N$ *such that*

$$a_2^i | a_1 \sim \sum_{j=1}^{N} \frac{B^{ij} w_1^j}{\sum_l B^{il} w_1^l} \delta_{a_1^j}(\cdot) \text{ and } w_2 = B w_1^T.$$

Then we have

$$\mathbb{P}(a_2^i = j) = (BA)^{ij} w_0^j / \sum_k (BA)^{ik} w_0^k \text{ and } w_2 = BA w_0^T.$$

Proof. Writing $\mathbb{P}(a_2^i = j) = \sum_k \mathbb{P}\{a_2^i = a_1^k\} \mathbb{P}(a_1^k = j)$ and putting

$$\mathbb{P}(a_2^i = a_1^k) = \frac{B^{ik} w_1^k}{\sum_l B^{il} w_1^l},\ \mathbb{P}(a_1^k = j) = \frac{A^{kj} w_0^j}{\sum_l A^{kl} w_0^l},\ w_1^k = \sum_l A^{kl} w_0^l$$

and simplifying we get

$$\mathbb{P}(a_2^i = j) = \sum_k \frac{B^{ik} A^{kj} w_0^j}{\sum_l \sum_p B^{il} A^{lp} w_0^p} = \frac{(BA)^{ij} w_0^j}{\sum_p (BA)^{ip} w_0^p}.$$

□

Recall that, in the graph of interaction structure of a resampling algorithm we identified the edges by non-zero probabilities. Roughly speaking, proposition 3.1 states that when we stack up the interaction graphs identified by two matrices A and B, paths from the top row to the bottom row are characterized by non-zero entries of the product matrix BA. Moreover, this strategy results in the same conditional probabilities on

each output particle given the input particles as we would get if we had performed αSMC resampling using BA matrix.

An approach towards the goal of generalizing multinomial resampling algorithm to get a more parallelizable resampling algorithm is to consider the m-fold factorizations of the matrix $\mathbf{1}_N$ into sparse doubly stochastic matrices. Letting $\mathbf{1}_N = \prod_{k=1}^{m} A_k$ where $A_1, \ldots, A_m$ are doubly stochastic matrices, augmented resampling algorithm has the lack-of-bias property and moment bound stated in the next theorem taken from [51]:

Theorem 3.2. *Consider Figure 3.3., with $w_{in}^i = g(x_{in}^i)$ where $g : \mathbb{X} \to \mathbb{R}_+$ is bounded. Fix $m \geq 1$ and suppose that $(A_k)_{k=1}^m$ are doubly stochastic matrices and $\prod_{k=1}^{m} A_k = \mathbf{1}_N$. Then for any bounded test function $\varphi : \mathbb{X} \to \mathbb{R}$, we have*

$$\mathbb{E}\left(\frac{1}{N}\sum_{i=1}^{N}\varphi(x_{out}^i)\middle| x_{in}\right) = \frac{\sum_i w_{in}^i \varphi(x_{in}^i)}{\sum_i w_{in}^i}, \tag{3.2}$$

and for any $p \geq 1$ there exists a finite constant b_p, depending only on p, such that no matter what the distribution of $(x_{in}^i)_{i=1}^N$ is,

$$\mathbb{E}\left(\left|\left(\frac{1}{N}\sum_i w_{in}^i\right)\left(\frac{1}{N}\sum_i \varphi(x_{out}^i)\right) - \frac{1}{N}\sum_i w_{in}^i\varphi(x_{in}^i)\right|^p\right) \leq b_p\left(\frac{m}{N}\right)^{\frac{p}{2}} C_{g,\varphi}^p, \tag{3.3}$$

where $C_{g,\varphi} = \|g\|_\infty \sup_{x,y\in\mathbb{X}} |\varphi(x) - \varphi(y)|$.

Proof of this theorem, along with other results on convergence properties of butterfly resampling is presented in [51]. While lack-of-bias property 3.2 follows from the direct computation of expectation using marginal probabilities given in 3.1, moment bound in 3.3 can be shown using martingale theory.

As in αSMC resampling, efficiently parallelizable instances of augmented resampling algorithm can be obtained by considering special forms of stochastic matrices to employ a data reuse approach. In the next section we present the *butterfly resampling* algorithm, a special case of augmented resampling algorithm that use a parameterized family of stochastic matrices to induce an interaction structure characterized by

modular congruence relations on particle indices.

3.2. Butterfly Resampling

Let $N = r_1 r_2 \dots r_m$ be a m-fold factorization of N with $\min_k\{r_k\} \geq 2$ and let $\mathbb{A}_N^m = \{A_1, \dots, A_m\}$ be a family of doubly stochastic matrices that are defined as follows:

$$A_k = I_{r_m} \otimes \cdots \otimes I_{r_{k+1}} \otimes \mathbf{1}_{r_k} \otimes I_{r_{k-1}} \otimes \cdots \otimes I_{r_1} = I_{r_{k+1} \dots r_m} \otimes \mathbf{1}_{r_k} \otimes I_{r_1 \dots r_{k-1}}. \quad (3.4)$$

An example class of matrices parameterized in this way with $N = 8$ and $r_k = 2$ for $k = 1, 2, 3$ and their pairwise products are depicted in Figure 3.4. Product of the matrices in $\mathbb{A}_N^m$ in any order gives us the matrix $\mathbf{1}_N$. In the following proposition we give certain useful properties of the family of matrices $\mathbb{A}_N^m$:

Proposition 3.3. *For the family of matrices $\mathbb{A}_N^m = \{A_1, \dots, A_m\}$ where each A_k, $k = 1, \dots, m$ defined as in Equation 3.4 we have the following properties:*

(i) For any $k = 1, \dots, m$ we have $A_k^2 = A_k$, i.e. they are idempotent,

(ii) For any $k, l = 1, \dots, m$ we have $A_k A_l = A_l A_k$, i.e. $\mathbb{A}_N^m$ is a commutative family of matrices,

(iii) For any permutation of $\{1, \dots, m\}$, say $\sigma : \{1, \dots, m\} \to \{1, \dots, m\}$, we have $\prod_{k=1}^m A_{\sigma(k)} = \mathbf{1}_N$

Proof. Recall that for any matrices A_1, A_2, B_1, B_2 with compatible dimensions, Kronecker product satisfies the mixed product property : $(A_1 \otimes B_1)(A_2 \otimes B_2) = (A_1 A_2) \otimes (B_1 B_2)$. Then by associativity of Kronecker product and matrix product and mixed product property, for any finite family of matrices $\{A_{ij}\}_{i,j=1}^{p,q}$ with compatible dimensions we have

$$\prod_{i=1}^{p} (A_{i1} \otimes \cdots \otimes A_{iq}) = (\prod_{i=1}^{p} A_{i1}) \otimes \cdots \otimes (\prod_{i=1}^{p} A_{iq}). \quad (3.5)$$

(i) By definition of $\mathbb{A}_N^m$ and Equation 3.5 with $p = 2, q = m$, we have for any $k = 1, \ldots, m$

$$\begin{aligned} A_k^2 =& (I_{r_m} \otimes \cdots \otimes I_{r_{k+1}} \otimes \mathbf{1}_{r_k} \otimes I_{r_{k-1}} \otimes \cdots \otimes I_{r_1}) \\ & \times (I_{r_m} \otimes \cdots \otimes I_{r_{k+1}} \otimes \mathbf{1}_{r_k} \otimes I_{r_{k-1}} \otimes \cdots \otimes I_{r_1}) \\ =& I_{r_m} \otimes \cdots \otimes I_{r_{k+1}} \otimes \mathbf{1}_{r_k} \otimes I_{r_{k-1}} \otimes \cdots \otimes I_{r_1} = A_k. \end{aligned}$$

(ii) By definition of $\mathbb{A}_N^m$ and Equation 3.5 with $p = 2, q = m$, we have for any $1 \leq k < l \leq m$

$$\begin{aligned} A_k A_l =& (I_{r_m} \otimes \cdots \otimes I_{r_{k+1}} \otimes \mathbf{1}_{r_k} \otimes I_{r_{k-1}} \otimes \cdots \otimes I_{r_1}) \\ & \times (I_{r_m} \otimes \cdots \otimes I_{r_{l+1}} \otimes \mathbf{1}_{r_l} \otimes I_{r_{l-1}} \otimes \cdots \otimes I_{r_1}) \\ =& I_{r_m} \otimes \cdots \otimes \mathbf{1}_{r_l} \otimes \cdots \otimes \mathbf{1}_{r_k} \otimes \cdots \otimes I_{r_1} \\ =& (I_{r_m} \otimes \cdots \otimes I_{r_{l+1}} \otimes \mathbf{1}_{l_k} \otimes I_{r_{l-1}} \otimes \cdots \otimes I_{r_1}) \\ & \times (I_{r_m} \otimes \cdots \otimes I_{r_{k+1}} \otimes \mathbf{1}_{r_k} \otimes I_{r_{k-1}} \otimes \cdots \otimes I_{r_1}) \\ =& A_l A_k \end{aligned}$$

(iii) By definition of $\mathbb{A}_N^m$ and Equation 3.5 with $p = m, q = m$, we have for any permutation σ of $\{1, \ldots, m\}$

$$\begin{aligned} \prod_{k=1}^{m} A_{\sigma(k)} &= \prod_{k=1}^{m} (I_{r_m} \otimes \cdots \otimes \mathbf{1}_{r_k} \otimes \cdots \otimes I_{r_1}) \\ &= (\mathbf{1}_{r_m} \prod_{i=1}^{m-1} I_{r_m}) \otimes \cdots \otimes (\mathbf{1}_{r_1} \prod_{i=1}^{m-1} I_{r_1}) = \mathbf{1}_{r_m} \otimes \cdots \otimes \mathbf{1}_{r_1} = \mathbf{1}_N. \end{aligned}$$

□

We will refer to the instance of augmented resampling algorithm that uses this family of matrices $\mathbb{A}_N^m$ as *butterfly resampling* algorithm with reference to resemblance of its interaction structure to the butterfly diagram encountered in Cooley-Tukey algorithm for FFT computation [61] and the sequence of factors $r = (r_1, \ldots, r_m)$ as the

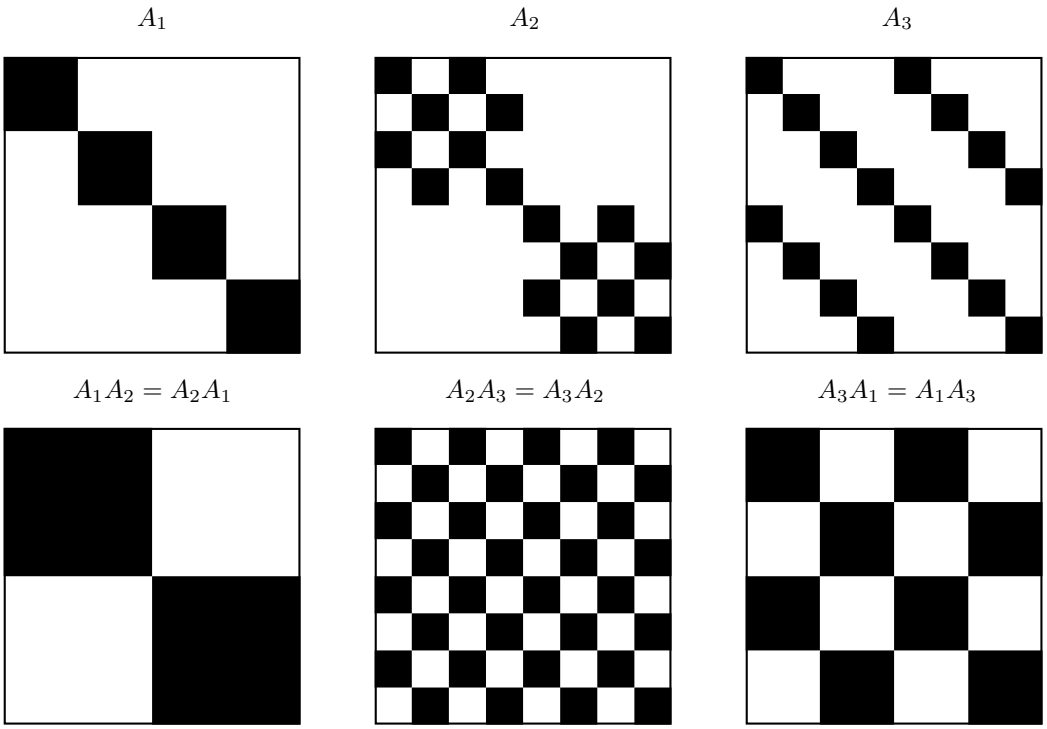


Figure 3.4. A matrices and their pairwise products for $m = 3$, $r_1, r_2, r_3 = 2$, zero entries are shown in white, nonzero entries are shown in black.

radix sequence with the same reference. The interaction structure of butterfly resampling algorithm parameterized with $N = 8$, $m = 3$ and $r_1, r_2, r_3 = 2$ is depicted in Figure 3.5.

What A_k matrices do, effectively, is to form N/r_k particle blocks of size r_k each, then within each block, a standard multinomial resampling operation can be performed independently from other blocks in parallel. This kind of interaction structure fits better into local communication structure of GPU threads which we described in Section 1.4. At the next stage, using another A_k matrix, we form new blocks in a way that each block comprises of particles that did not interact previously. This way, we increase the overall interaction at each stage while keeping the degree of *stage-wise* interaction below a threshold.

Note that, at each stage k, the matrix A_k induces an equivalence relation on the index set $\{1, \ldots, N\}$ characterized by nonzero entries of the matrix A_k as follows: $i \sim j \iff A_k^{ij} = A_k^{ji} > 0$. Nonzero entries of the matrix A_k further can be characterized

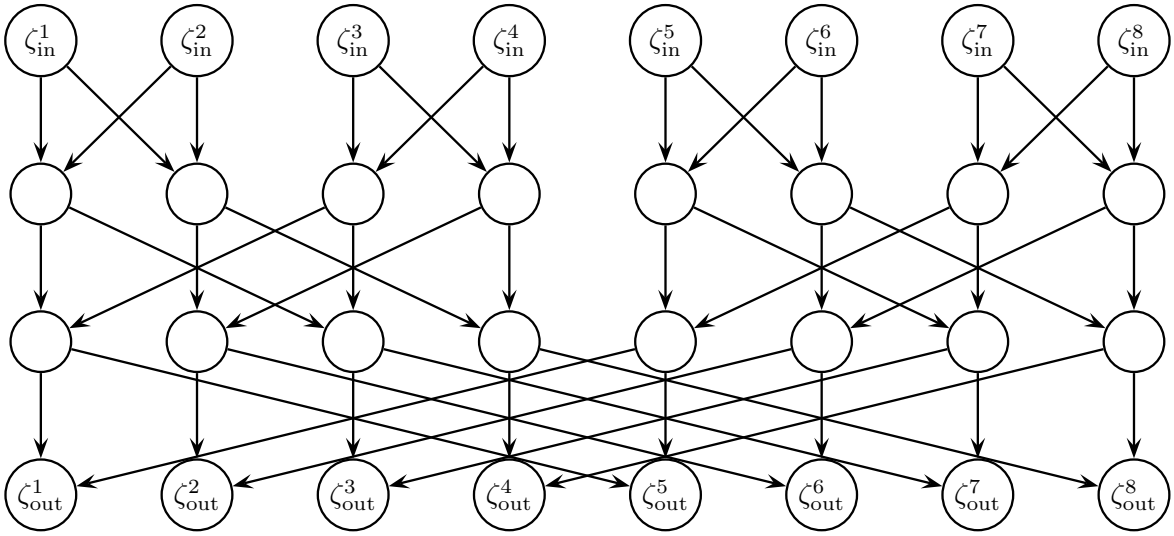


Figure 3.5. Interaction structure of butterfly resampling.

by modular congruence relations as follows:

$$A_k^{ij} = A_k^{ji} > 0 \iff \lfloor \tfrac{i-1}{r_1 \dots r_k} \rfloor = \lfloor \tfrac{j-1}{r_1 \dots r_k} \rfloor \text{ and } (i-1) \bmod (r_1 \dots r_{k-1}) = (j-1) \bmod (r_1 \dots r_{k-1}).$$

For a given radix r_k, partitioning of the index set $\{1, \dots, N\}$ into equivalence classes $([i_{r,k,p}])_{p=1}^{N/r_k}$ which are represented by minimal elements

$$i_{r,k,p} = \lfloor \tfrac{p-1}{r_1 \dots r_{k-1}} \rfloor r_1 \dots r_k + (p-1) \bmod (r_1 \dots r_{k-1}) + 1 \tag{3.6}$$

for $p = 1, \dots, N/r_k$, corresponds to above mentioned particle blocks structure. Within this block structure, block index of the ith particle in the kth stage is

$$\mathcal{I}_{r,k,i} = \lfloor \tfrac{i-1}{r_1 \dots r_k} \rfloor r_1 \dots r_{k-1} + (i-1) \bmod (r_1 \dots r_{k-1}) + 1. \tag{3.7}$$

In Figure 3.6. we give the pseudocode for butterfly multinomial resampling. Here we make the data reuse approach employed by butterfly resampling scheme more explicit: note that, in the parallel for block starting at line 4 we compute N/r_k non-overlapping prefix sums each involving r_k elements, so at the kth stage of butterfly resampling algorithm, connected components of the interaction structure comprise of

r_k input and r_k output particles for each. This allows us to control the degree of interaction at each stage by adjusting the maximum radix in r while maintaining full interaction at the end by considering factorizations of $\mathbf{1}_N$.

function ButterflyResample(w_{in}, x_{in}, $r = (r_k)_{k=1}^m$)
  $w_0 \leftarrow w_{\text{in}}$ and $\boldsymbol{\xi}_0 \leftarrow \mathbf{x}_{\text{in}}$
  for $k \in \{1, \ldots, m\}$ **do**
    for each $p \in \{1, \ldots, N/r_k\}$ **do**
      $W_{k,p} \leftarrow \text{PrefixSum}(w_{k-1}^{[i_{r,k,p}]})$ //Prefix sum for each block independently
    end for
    for each $i \in \{1, \ldots, N\}$ **do**
      $j^i \leftarrow \text{InversionSampling}(W_{k,\mathcal{I}_{r,k,i}}, [i])$ //Sample ancestors independently
      $\xi_k^i = \xi_{k-1}^{j^i}$ and $w_k^i \leftarrow W_{k,\mathcal{I}_{r,k,i}}^{r_k}/r_k$ //Update particles and weights
    end for
  end for
  $w_{\text{out}} \leftarrow w_m, x_{\text{out}} \leftarrow \xi_m$
  return $w_{\text{out}}, x_{\text{out}}$
end function

Figure 3.6. GPU Implementation of Butterfly Multinomial Resampling.

The implementation of Figure 3.6. involves m kernel calls for prefix sum computations, m kernel calls for sampling part and one kernel call for the update part. As in the multinomial resampling, asymptotic analysis of Figure 3.6. shows that it takes $\mathcal{O}(\log N)$ time to perform butterfly resampling algorithm. However, with detailed analysis we see the trade-off between using shared memory and kernel call overhead. At the kth stage of butterfly resampling, we have $\mathcal{O}(t_S \log_2 r_k)$ time cost associated to the binary searches in the sampling part. Summing this over all stages, it takes $\mathcal{O}(t_S \log_2 N + m t_K)$ time to perform butterfly resampling on the GPU, where t_K is the time it takes to launch the sampling kernel. Comparing this to $\mathcal{O}(t_G \log_2 N + t_K)$ time cost of standard multinomial resampling, we see that we are better off in terms

of memory access times since in general t_S is a much smaller constant than t_G, but we have more kernel call overhead. In practice, we keep the parameter m in the order of $\log_{r_{\max}} N$, so the cost associated to sampling part of the butterfly resampling is $\mathcal{O}(t_S \log_2 N + t_K \log_{r_{\max}} N)$. Since $\log_{r_{\max}} N$ grows slower than $\log_2 N$, we expect to see a speed improvement over standard multinomial resampling. The computation of prefix sum in butterfly resampling scheme have $\mathcal{O}(t_S \log_2 N + t_K \log_{r_{\max}} N)$ time cost as in multinomial resampling: PrefixSum function at the kth stage of butterfly takes $\mathcal{O}(t_S \log_2 r_k)$ time and summing over all stages we get the total time cost of $\mathcal{O}(t_S \log_2 N + t_K \log_{r_{\max}} N)$ associated to the prefix sum operation.

The local interaction structure of butterfly resampling algorithm, enables in-place propagation of particles in a particle filter context. Since threads within a thread block can perform barrier synchronization, copying of particles after resampling at each stage does not suffer from race conditions as in classical resampling algorithms. However, if the particles are very high dimensional large number of memory movements for copying the content of particles from one location to local buffer and from local buffer to another location can be demanding considering the multiple resampling stages of the butterfly resampling scheme. In such a scenario, we can wait until the butterfly resampling algorithm delivers an ancestors vector and then use a double buffer strategy as in the case of multinomial resampling. We present the implementations of in-place propagation of particles for butterfly resampling scheme in Appendix A.3.

Pay-off of more parallelizable interaction structure of butterfly resampling algorithm manifests itself in the statistical efficiency. Recall from Theorem 2.1 that, as we increase the number of particles N to the infinity, asymptotic fluctuations of the particle approximation errors of the particle filter that employs standard multinomial resampling scales with $\sqrt{N}$, a consequence of conditional independence of output particles given input particles. However, we eliminate this conditional independence structure by augmenting additional layers and we have $\mathbb{P}(a^{i_1} = j_1, a^{i_2} = j_2) \neq \mathbb{P}(a^{i_1} = j_1)\mathbb{P}(a^{i_2} = j_2)$, so the order of the asymptotic fluctuations of approximation errors also change. Analysis of asymptotic error fluctuations for the most general form of butterfly resampling is more involved than usual analysis of resampling algorithms and

requires combinatorial analysis of conditional independence graphs. Here, we present convergence and central limit theorems for two special cases of butterfly resampling algorithm for which the proofs are given in [51].

In the *radix-r butterfly resampling* algorithm, we have a constant radix sequence where $r_k = r$ for all $k = 1, \ldots, m$ with $N = r^m$. So, as we increase the number of layers m, we increase the number of particle N but the maximum degree of interaction at each stage $k = 1, \ldots, m$ is kept fixed. For the particle filter algorithm with radix-r butterfly resampling we have the following convergence and central limit theorem:

Theorem 3.4. *For any bounded test function $\varphi : \mathbb{X} \to \mathbb{R}$,$r \geq 2$ and $n \geq 1$, define*

$$\begin{aligned}
\sigma^2_{R,0}(\varphi, r) &:= \pi((\varphi - \pi_0(\varphi))^2), \\
\sigma^2_{R,n}(\varphi, r) &:= \hat{\sigma}^2_{R,n-1}(f(\varphi), r), \\
\hat{\sigma}^2_{R,0}(\varphi, r) &:= (1 - r^{-1})\hat{\pi}_0((\varphi - \pi_0(\varphi))^2), \\
\hat{\sigma}^2_{R,n}(\varphi, r) &:= (1 - r^{-1})\hat{\pi}_n((\varphi - \pi_0(\varphi))^2) + \pi_n(g_n)^{-2}\sigma^2_{R,n}(g_n(\varphi - \hat{\pi}_n(\varphi)), r).
\end{aligned}$$

Then, the particle filter with radix-r butterfly resampling has the properties that

$$\begin{aligned}
&\pi_0^N(\varphi) - \pi_0(\varphi) \xrightarrow{a.s.} 0 \text{ and } \sqrt{N}(\pi_0^N(\varphi) - \pi_0(\varphi)) \xrightarrow{dist.} \mathcal{N}(0, \sigma^2_{R,0}(\varphi, r)), \\
&\hat{\pi}_0^N(\varphi) - \hat{\pi}_0(\varphi) \xrightarrow{a.s.} 0 \text{ and } \sqrt{\frac{N}{\log_r N}}(\hat{\pi}_0^N(\varphi) - \hat{\pi}_0(\varphi)) \xrightarrow{dist.} \mathcal{N}(0, \hat{\sigma}^2_{R,0}(\varphi, r)),
\end{aligned}$$

and for any $n \geq 1$,

$$\begin{aligned}
&\pi_n^N(\varphi) - \pi_n(\varphi) \xrightarrow{a.s.} 0 \text{ and } \sqrt{\frac{N}{\log_r N}}(\pi_n^N(\varphi) - \pi_n(\varphi)) \xrightarrow{dist.} \mathcal{N}(0, \sigma^2_{R,n}(\varphi, r)), \\
&\hat{\pi}_n^N(\varphi) - \hat{\pi}_n(\varphi) \xrightarrow{a.s.} 0 \text{ and } \sqrt{\frac{N}{\log_r N}}(\hat{\pi}_n^N(\varphi) - \hat{\pi}_n(\varphi)) \xrightarrow{dist.} \mathcal{N}(0, \hat{\sigma}^2_{R,n}(\varphi, r)),
\end{aligned}$$

where the convergence is as $N \to \infty$ along the sequence of integer power of r, i.e. $N = r^m$, $m = 1, 2, \ldots$ for which the radix-r butterfy resampling algorithm is defined.

From Theorem 3.4, we infer that limiting the interaction in order that the degree of each vertex in the interaction graph of resampling does not grow with N cause approximation errors of the particle filter decay slower than that of standard multinomial resampling. To prevent this effect we need to come up with another special form of butterfly resampling algorithm with better scaling behaviour.

In the *mixed radix-r butterfly resampling* algorithm, we have $m = 2$, a constant first radix $r_1 = r$ and a second radix $r_2 = c$ where $N = rc$. Here, we keep the number of stages fixed at two while increasing the number of particles N by adjusting the second radix c. For the particle filter algorithm with mixed radix-r butterfly resampling algorithm we have the following convergence and central limit theorem:

Theorem 3.5. *For any bounded test function $\varphi : \mathbb{X} \to \mathbb{R}$ and $r \geq 2$, define*

$$\begin{aligned}
\sigma^2_{M,0}(\varphi, r) &:= \pi((\varphi - \pi_0(\varphi))^2), \\
\sigma^2_{M,n}(\varphi, r) &:= \hat{\sigma}^2_{M,n-1}(f(\varphi), r) + \hat{\pi}(f((\varphi - f(\varphi))^2)), n \geq 1 \\
\hat{\sigma}^2_{M,n}(\varphi, r) &:= (2 - r^{-1})\hat{\pi}_n((\varphi - \pi_0(\varphi))^2) + \pi_n(g_n)^{-2}\sigma^2_{M,n}(g_n(\varphi - \hat{\pi}_n(\varphi)), r), n \geq 0.
\end{aligned}$$

Then, for $n \geq 0$ the particle filter with radix-r butterfly resampling has the properties that

$$\begin{aligned}
\pi^N_n(\varphi) - \pi_n(\varphi) &\xrightarrow{a.s.} 0 \text{ and } \sqrt{N}(\pi^N_n(\varphi) - \pi_n(\varphi)) \xrightarrow{dist.} \mathcal{N}(0, \sigma^2_{M,n}(\varphi, r)), \\
\hat{\pi}^N_n(\varphi) - \hat{\pi}_n(\varphi) &\xrightarrow{a.s.} 0 \text{ and } \sqrt{N}(\hat{\pi}^N_n(\varphi) - \hat{\pi}_n(\varphi)) \xrightarrow{dist.} \mathcal{N}(0, \hat{\sigma}^2_{M,n}(\varphi, r)),
\end{aligned}$$

where the convergence is as $N \to \infty$ along the sequence of integer multiples of r, i.e. $N = rc$, $c = 1, 2, \ldots$ for which the mixed radix-r butterfy resampling algorithm is defined.

From Theorem 3.5, we infer that by letting the maximum degree of vertices in the interaction graph of resampling algorithm grow with N we can keep the scaling of the approximation error at the same level as the standard multinomial resampling, however with a greater asymptotic variance.

So far, we have considered the butterfly counterpart of the multinomial resampling algorithm, but butterfly resampling mechanism can be applied to stratified and systematic resampling algorithms to obtain generalizations of them just with a small modification. Recall that within each block, we perform multinomial resampling to r_k particles at the kth stage of butterfly resampling algorithm. Instead of multinomial resampling, we can perform stratified or systematic resampling to decrease the variance. In addition to decrease in variance, this would save us from the cost of binary search used in inversion sampling procedure. The time costs associated to butterfly stratified and butterfly systematic resampling are the same up to a constant factor. At the kth stage of butterfly stratified or systematic resampling, we have $r_k \max_i \left\{ \frac{w_k^i}{W_{k,\mathcal{I}_{r,k,i}}^{r_k}} \right\} t_S$ average time cost associated to the **for** loop part of the conversion of cumulative offspring vectors to the ancestors vectors. Since we have $\max_i \left\{ \frac{w_k^i}{W_{k,\mathcal{I}_{r,k,i}}^{r_k}} \right\} < 1$, for all $k = 1, \ldots, m$, we have a total of $\mathcal{O}(c' t_S + t_K \log_{r_{\max}} N)$ time cost associated to the sampling part of butterfly stratified and systematic resampling algorithms, where c' is a constant depending on the potential function g used for determining weights and the radix sequence r. Comparing this to $\mathcal{O}(ct_G + t_K)$ time cost of sampling parts of standard stratified and systematic resampling algorithms, we see that dominating term of $t_K \log_{r_{\max}} N$ of butterfly resampling algorithm to be disadvantageous. Time cost associated to the prefix sum is $\mathcal{O}(t_S \log_2 N)$ as in the other algorithms, since we have changed only the process of sampling ancestors.

We generalized the constrained interaction structure of αSMC resampling by considering certain parameterized factorizations of the matrix $\mathbf{1}_N$ which characterizes the interaction structure of standard multinomial resampling algorithm. Our next step is to introduce *adaptivity* of αSMC resampling into our augmented resampling framework.

3.3. Adaptive Butterfly Resampling

A practical extension of butterfly resampling scheme can be obtained by introducing the idea of adaptive resampling into our augmented resampling framework. Recall that, in αSMC resampling, degree of interaction was adjusted according to some func-

tional of particle-weight set at the hand. In Figure 3.7. we introduce *adaptive butterfly resampling* in which overall interaction is adjusted to maintain an ESS bound instead of performing all m stages of butterfly resampling as in Figure 3.6. Again, here we make the parallelization scheme explicit at line 4 by considering the particle blocks formed by the A_k matrices. In this pseudocode, we employ multinomial resampling within each block to sample the ancestors. However, as in the case of full butterfly resampling scheme of previous section, this scheme can be adapted to stratified and systematic resampling algorithms as well, to generalize their adaptive versions.

1: **function** AdaptiveButterflyResample(w_{in}, x_{in}, $r = (r_k)_{k=1}^m$,τ)
2: $w_0 \leftarrow w_{\text{in}}, \xi_0 \leftarrow x_{\text{in}}$ and $k \leftarrow 1$
3: **while** $\mathcal{E}^N(w_{k-1}) < \tau$ **do** //While ESS is below threshold
4: **for each** $p \in \{1, \ldots, N/r_k\}$ **do**
5: $W_p \leftarrow \text{PrefixSum}(w_{k-1}^{[i_{r,k,p}]})$ //Prefix sum for each block independently
6: **end for**
7: **for each** $i \in \{1, \ldots, N\}$ **do**
8: $j^i \leftarrow \text{InversionSampling}(W_{\mathcal{I}_{r,k,i}}, [i])$ //Sample ancestors independently
9: $\xi_k^i = \xi_{k-1}^{j^i}$ and $w_k^i \leftarrow W_{\mathcal{I}_{r,k,i}}^{r_k}/r_k$ //Update particles and weights
10: **end for**
11: $k \leftarrow k + 1$
12: **end while**
13: $w_{\text{out}} \leftarrow w_{k-1}, x_{\text{out}} \leftarrow \xi_{k-1}$
14: **return** $w_{\text{out}}, x_{\text{out}}$
15: **end function**

Figure 3.7. GPU Implementation of Adaptive Butterfly Resampling.

Adaptation scheme of Figure 3.7. requires the computation of ESS at every stage of butterfly resampling algorithm. However, in general, cost of ESS computation is not greater than resampling operation and by doing away with some stages of butterfly resampling we can gain some speed-up. Furthermore, in practice ESS computation

can be overlapped with the computation of cumulative sums so as to cut down the kernel call overheads in GPUs. Nonetheless, relative performance of particle filter employing adaptive resampling scheme instead of full resampling scheme depends very much on the specific distributions of the underlying state-space model and proposal distributions of the particle filter. When working with a model in which adaptive resampling scheme tends to perform nearly all resampling stages at every time step, overhead of the ESS computation may exceed the gain of not performing remaining few resampling stages.

4. EXPERIMENTS AND RESULTS

In this section we provide our empirical results to support our theoretical results and demonstrate the performances of classical and butterfly resampling algorithms in GPU implementations of particle filters. We implemented classical and butterfly resampling algorithms separately in CUDA platform using CUDA Toolkit version 5.5. In our implementations, each particle is mapped to a CUDA thread. In order to fit the abstract graph structure of butterfly resampling to GPU architecture, we consider radix sequences with radices less than the maximum number of threads per block, which is 1024 threads in our GPU. In our experiments, we used NVIDIA GeForce GTX680 GPU with CUDA capability 3.0.

We conduct experiments under three main categories: single step of butterfly resampling, comparison of resampling classical and butterfly resampling schemes in a simple particle filter context and comparison of resampling schemes in a practical application. In Section 4.1, we consider the butterfly resampling scheme in isolation from a particle filter context. First, we describe the general setting of our experiments and how we compare the Monte Carlo error associated to the resampling algorithms in Section 4.1.1, then to verify the theoretical results concerning the butterfly resampling algorithm and obtain practical guidelines for the selection of radix sequences we observe the behaviour of Monte Carlo error under different radix sequence regimes in subsections 4.1.2 and 4.1.3. Finally, in Section 4.1.4 we compare the performance of single steps of classical and butterfly resampling algorithms in terms of speed and Monte Carlo error.

In Section 4.2 we compare the classical and butterfly resampling schemes in terms of speed and Monte Carlo error in two different simple particle filtering scenarios: in Section 4.2.2 the bootstrap particle filter in which resampling is performed in each timestep of particle filter until the weights are uniform and in Section 4.2.3 the adaptive resampling particle filter in which resampling is performed until an ESS threshold is achieved. Finally, in Section 4.3 we present performance comparison of resampling

under full interaction and constrained interaction in a more practical context by considering SMCEM algorithm, a parameter estimation algorithm that uses particle filter as a subroutine.

4.1. Single Step of Butterfly Resampling

In this section we investigate theoretical and practical aspects of butterfly resampling by considering one step of resampling in isolation from a particle filter context. In Section 4.1.1, we describe the setting in which we evaluate the Monte Carlo error of resampling algorithms. In Section 4.1.2, we verify the theoretical results concerning the Monte Carlo error distribution of butterfly resampling by simulation. In Section 4.1.3, we investigate the effect of choice of radix sequence on Monte Carlo error empirically, by considering a fixed number of particles scenario. In Section 4.1.4, we compare performances of standard multinomial resampling and butterfly resampling algorithms in terms of Monte Carlo error and speed.

4.1.1. General Setting

In our general setting, we have N input random variables $\xi_0 = (\xi_0^i)_{i=1}^N$ i.i.d. from a sampling distribution π_0 and associated weights computed by a potential function g such that $w_0^i = g(\xi_0^i)$. We are interested in the distribution

$$\hat{\pi}_0(dx) := \pi_0(dx)g(x)/\pi_0(g),$$

more specifically, for a certain test function φ, we wish to estimate the expectation of φ w.r.t. $\hat{\pi}_0$ that we denote with $\hat{\pi}_0(\varphi)$.

We consider the butterfly resampling algorithm parametrized by a radix sequence $r = (r_1, \ldots, r_m)$ such that $\prod_{k=1}^m r_k = N$ and write $\xi_k = (\xi_k^i)_{i=1}^N$ for the random output particles and $w_k = (w_k^i)_{i=1}^N$ for the output weights of the kth stage of the butterfly resampling algorithm. Note that multinomial resampling algorithm is a special case of butterfly resampling with $m = 1$ and $r_1 = N$. For notational compactness, in

the definitions of quantities given below, we suppress the dependence on the particular resampling algorithm, its parameterization, particle sampling distribution π_0 and potential function g.

We compare the performance of resampling algorithms in terms of Monte Carlo error measured by estimation errors. For a given number of particles N and a test function φ, our focus is on the estimation error :

$$E_\varphi^N = \frac{1}{N}\sum_{i=1}^{N}\varphi(\xi_m^i) - \hat{\pi}_0(\varphi), \tag{4.1}$$

which is a random quantity. Note that, this error term is in fact sum of two errors, one stemming from the resampling procedure and other one stemming from importance sampling procedure:

$$E_\varphi^N = \left(\frac{1}{N}\sum_{i=1}^{N}\varphi(\xi_m^i) - \frac{\sum_{i=1}^{N} g(\xi_0^i)\varphi(\xi_0^i)}{\sum_{i=1}^{N} g(\xi_0^i)}\right) + \left(\frac{\sum_{i=1}^{N} g(\xi_0^i)\varphi(\xi_0^i)}{\sum_{i=1}^{N} g(\xi_0^i)} - \hat{\pi}_0(\varphi)\right).$$

In our experiments, we estimate and compare the Mean Squared Error (MSE) of the sample averages taken over resampled particles defined as follows :

$$\mathrm{MSE}_\varphi^N = \mathbb{E}\left[\left(\frac{1}{N}\sum_{i=1}^{N}\varphi(\xi_m^i) - \hat{\pi}_0(\varphi)\right)^2\right] = \mathbb{E}[(E_\varphi^N)^2], \tag{4.2}$$

where the expectation is taken over different realizations of resampling algorithm and initial particle set.

We also consider another error measure that does not include the error of the estimates made by weighted samples, which we call *local error*. For a given number of particles N, a test function φ and a potential function g used for weight computation as above, we define the local error as follows:

$$LE_\varphi^N = \frac{1}{N}\sum_{i=1}^{N}\varphi(\xi_m^i) - \frac{\sum_{i=1}^{N} g(\xi_0^i)\varphi(\xi_0^i)}{\sum_{i=1}^{N} g(\xi_0^i)}. \tag{4.3}$$

Similar to MSE, we define local MSE as follows :

$$\mathrm{LMSE}_{\varphi}^{N} = \mathbb{E}\left[\left(\frac{1}{N}\sum_{i=1}^{N}\varphi(\xi_m^i) - \frac{\sum_{i=1}^{N} g(\xi_0^i)\varphi(\xi_0^i)}{\sum_{i=1}^{N} g(\xi_0^i)}\right)^2\right] = \mathbb{E}[(LE_{\varphi}^{N})^2], \tag{4.4}$$

where the expectation is taken over different realizations of the resampling algorithm and initial particle set. We remark that, by using towering property of conditional expectations and Equation 3.2 of Theorem 3.2, we can decompose MSE as the sum of local MSE and MSE of importance sampling estimator of $\hat{\pi}_0(\varphi)$ as follows :

$$\begin{aligned}
\mathrm{MSE}_{\varphi}^{N} =& \mathbb{E}\left[\left(\frac{1}{N}\sum_{i=1}^{N}\varphi(\xi_m^i) - \hat{\pi}_0(\varphi)\right)^2\right] \\
=& \mathbb{E}\left[\left(\frac{1}{N}\sum_{i=1}^{N}\varphi(\xi_m^i) - \frac{\sum_{i=1}^{N} g(\xi_0^i)\varphi(\xi_0^i)}{\sum_{i=1}^{N} g(\xi_0^i)}\right)^2 + \left(\frac{\sum_{i=1}^{N} g(\xi_0^i)\varphi(\xi_0^i)}{\sum_{i=1}^{N} g(\xi_0^i)} - \hat{\pi}_0(\varphi)\right)^2\right] \\
=& \mathrm{LMSE}_{\varphi}^{N} + \mathbb{E}\left[\left(\frac{\sum_{i=1}^{N} g(\xi_0^i)\varphi(\xi_0^i)}{\sum_{i=1}^{N} g(\xi_0^i)} - \hat{\pi}_0(\varphi)\right)^2\right].
\end{aligned}$$

According to our theory, both $\mathrm{MSE}_{\varphi}^{N}$ and $\mathrm{LMSE}_{\varphi}^{N}$ should converge to the same quantity as $N \to \infty$, namely $\hat{\sigma}_0^2(\varphi, r) = (1-r^{-1})\hat{\pi}_0((\varphi - \hat{\pi}_0(\varphi))^2)$. However, above decomposition shows that we should expect $\mathrm{MSE}_{\varphi}^{N}$ to be higher than $\mathrm{LMSE}_{\varphi}^{N}$ for finite N, as we will see later in the experimental results.

In our experiments we estimate MSEs defined above by sampling from the error distribution. To estimate $\mathrm{MSE}_{\varphi}^{N}$, we sample M particle sets $(x_{0,j})_{j=1}^{M}$ independently, where particles within a particle set are i.i.d from the distribution π_0. We perform the resampling algorithm with inputs $(x_{0,j}, w_{0,j} = (w_{0,j}^i = g(x_{0,j}^i))_{i=1}^{N})_{j=1}^{M}$ to produce M ancestor vectors $(a_j = (a_j^i)_{i=1}^{N})_{j=1}^{M}$ such that $\xi_{m,j}^i = x_{0,j}^{a_t^i}$. Then estimated MSE is defined as :

$$\widehat{\mathrm{MSE}}_{\varphi}^{N,M} = \frac{1}{M}\sum_{j=1}^{M}\left(\frac{1}{N}\sum_{i=1}^{N}\varphi(x_{0,j}^{a_j^i}) - \hat{\pi}_0(\varphi)\right)^2 \tag{4.5}$$

When we do not have the value of $\hat{\pi}_0(\varphi)$, we can use estimates of local MSE. For this, we generate M particle-weight sets $(x_{0,j}, w_{0,j})$ and corresponding ancestor vectors $(a_j)_{j=1}^M$ as above. Then estimated local MSE is computed as follows:

$$\widehat{\mathrm{LMSE}}_{\varphi}^{N,M} = \frac{1}{M}\sum_{j=1}^{M}\left(\frac{1}{N}\sum_{i=1}^{N}\varphi(x_{0,j}^{a_j^i}) - \frac{\sum_{i=1}^{N} g(x_{0,j}^i)\varphi(x_{0,j}^i)}{\sum_{i=1}^{N} g(x_{0,j}^i)}\right)^2. \tag{4.6}$$

4.1.2. Error distribution and MSE as $N \to \infty$

In this section, our goal is to empirically verify and confirm the theoretical results related to the Monte Carlo error of the butterfly resampling algorithm. For this, we estimate the distribution of error associated to a test function, E_{φ}^N defined in Equation 4.1, and its local MSE, MSE_{φ}^N defined in Equation 4.4, as we increase the number of particles N. In Theorem 3.4, we stated that the error associated to estimates made by resampled particles, when scaled correctly, converges in distribution to a Gaussian random variable as we increase the number of particles while keeping the radix fixed. To verify these result we will compare the histograms and the local MSEs of the scaled empirical errors for different numbers of particles.

In our experiment we consider the fixed radix case with radix 2, i.e. the radix sequences of the form $(r_1, \ldots, r_m)$ with $r_k = 2$ for all $k = 1, \ldots, m$. For a fixed number of particles $N = 2^m$, we generate $T = 10000$ particle sets $\{\xi_{0,t}\}_{t=1}^T$ independently, where the particle sampling distribution π_0 is uniform over $[-2\sigma, 2\sigma]$ and we assign weights $(w_0^i)_{i=1}^N$ to particles using the potential function $g(x) = \exp\{-x^2/2\sigma^2\}$ with $\sigma = 10$. Then we run butterfly resampling for each particle-weight set to compute local MSE estimation $\widehat{\mathrm{LMSE}}_{\varphi}^N$ for the test function $\varphi(x) = x$. We repeat this procedure for particle numbers $N = 2, 2^2, \ldots, 2^{20}$.

Histograms of the errors E_{φ}^N scaled by $\sqrt{N/log_2 N}$ are given in Figure 4.1. Red lines in the same figure show the p.d.f. of the limiting Gaussian distribution scaled to fit the histograms. Estimated local MSEs, $\widehat{\mathrm{LMSE}}_{\varphi}^N$, scaled with $N/log_2 N$ are plotted in

Figure 4.2., while dashed red line shows the variance of the limiting normal distibution.

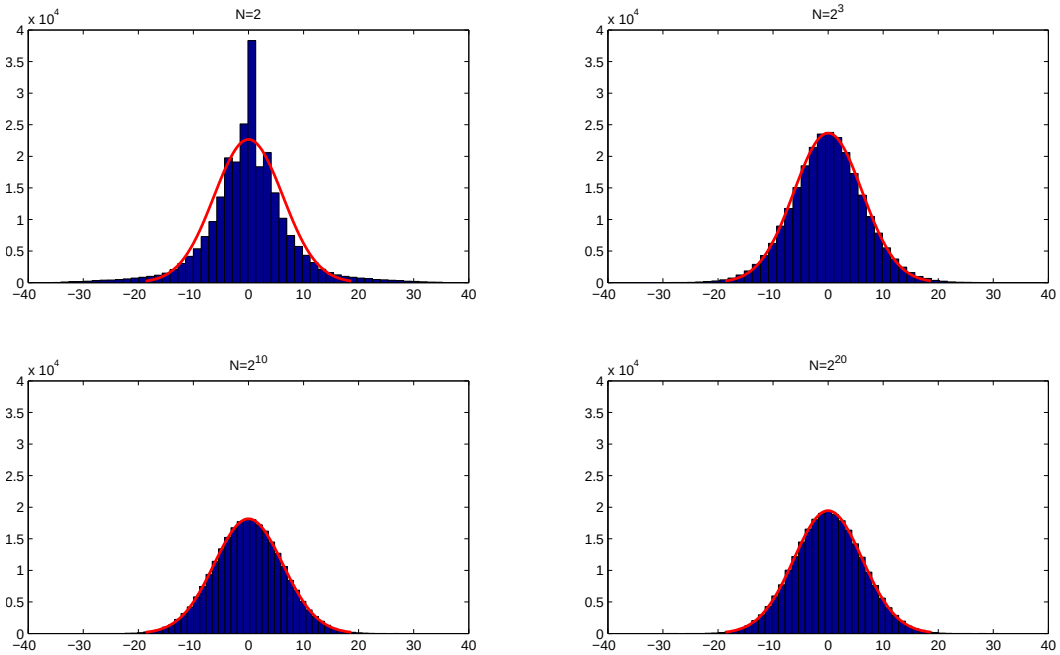


Figure 4.1. Histograms of the errors scaled with $\sqrt{N/log_2 N}$, red line shows the pdf of limiting error distribution.

4.1.3. Error in case of fixed number of particles

In this section, our goal is to empirically measure the effect of radix sequence on the Monte Carlo error of the butterfly resampling algorithm in a fixed number of particles regime, in order to obtain practical guidelines for implementation details. In particular, we perform two experiments to study how the error, as measured by MSE defined in Equation 4.2 varies by the choice of different radix sequences. We consider two different butterfly resampling schemata: (1) fixed radix/variable depth case and (2) mixed radix/fixed depth case. In both experiments, we first fix the number of particles and then choose the appropriate radix sequences.

In both experiments we use the same setup for particle-weight set generation : for a fixed number of particles N and radix sequence $(r_k)_{k=1}^m$, we generate $T = 1000$ particle sets $\{\xi_{0,t}\}_{t=1}^T$ independently, where the particle sampling distribution π_0 is uniform over $[-2\sigma, 2\sigma]$ and we assign weights $(w_0^i)_{i=1}^N$ to particles using the potential function $g(x) = \exp\{-x^2/2\sigma^2\}$ with $\sigma = 10$. Then we run butterfly resampling for

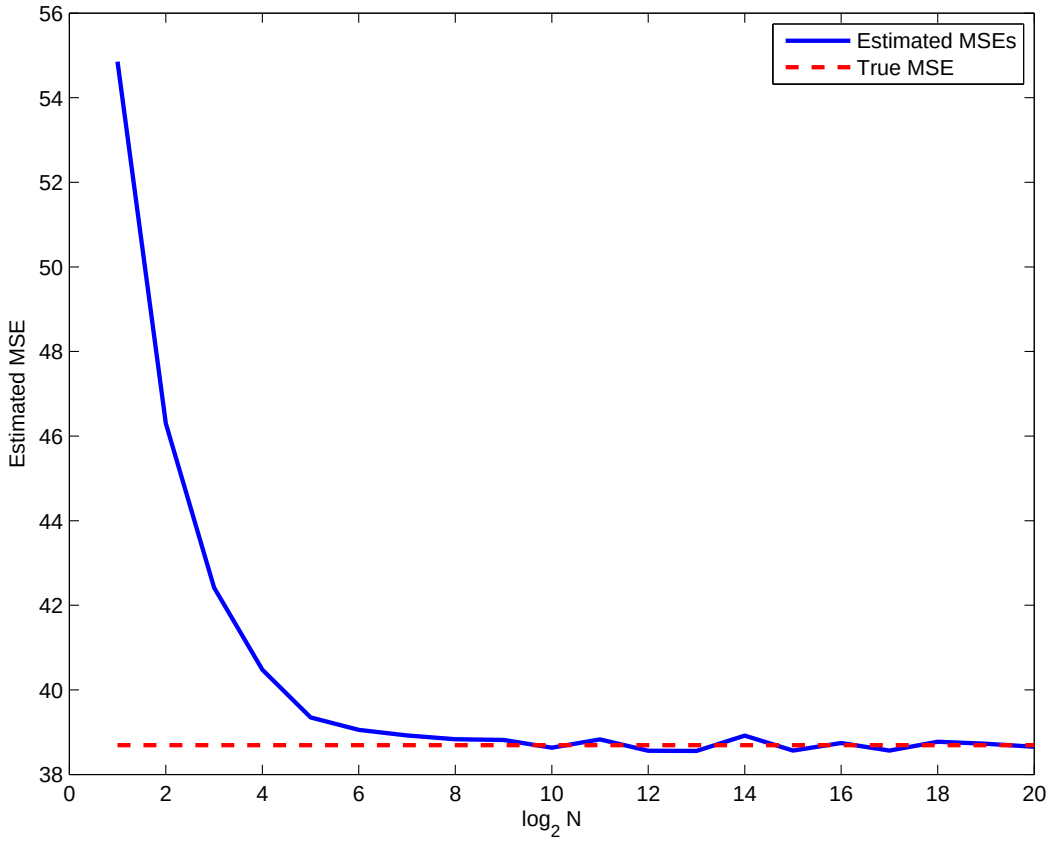


Figure 4.2. Plot of $\widehat{\text{LMSE}}_{\varphi}^{N}$ scaled with $N/log_2 N$ vs N.

each particle-weight set to compute local MSE estimate $\widehat{\text{LMSE}}_{\varphi}^{N}$ for the test function $\varphi(x) = x$.

In the first experiment, we choose the number of particles as $N = r^D$ where r is the fixed radix and D is the corresponding depth. We construct the radix sequences with different r and D by keeping N always constant. This generates radix sequences of the form $[r, r, \ldots, r]$ where r is repeated D times. We choose the number of particles as $N = 2^{18}$, as this is a sufficiently large number that allows us to consider a sufficiently broad range of radix sequences. In particular, we consider radices $r = 2, 2^2, 2^3, 2^6, 2^9, 2^{18}$, with corresponding depths as $D = 18, 9, 6, 3, 2, 1$, respectively. Results of this experiment are shown in Figure 4.3. The figure suggests that increasing the number of stages of butterfly resampling while keeping the number of particles fixed produce more MSE, so in practice it would be better in terms of Monte Carlo error to use butterfly resampling algorithm with smaller number of stages.

In the second experiment, we consider radix sequences with a fixed depth 2 and let the radix sequences vary as $\{r^d, r^{D-d}\}$ for $d = 1 \dots D-1$. In particular, we choose the radix sequences $(2, 2^{17}), (2^2, 2^{16}), \dots, (2^{16}, 2^2), (2^{17}, 2)$. Again we run the butterfly resampling algorithm by each radix sequence on all particles sets and plot the associated MSEs in the figure Figure 4.4. The figure suggests that changing the radix sequence while keeping the number of particle and number of stages fixed does not have substantial effect on MSE, so in practice it would not make a big difference in terms of Monte Carlo error to use any of the radix sequences with the same length. Combining this and the above result, in general we want to use radix sequences with smallest possible depth but once we fix the depth, we can decide which radix sequence to use according to other factors, such as speed.

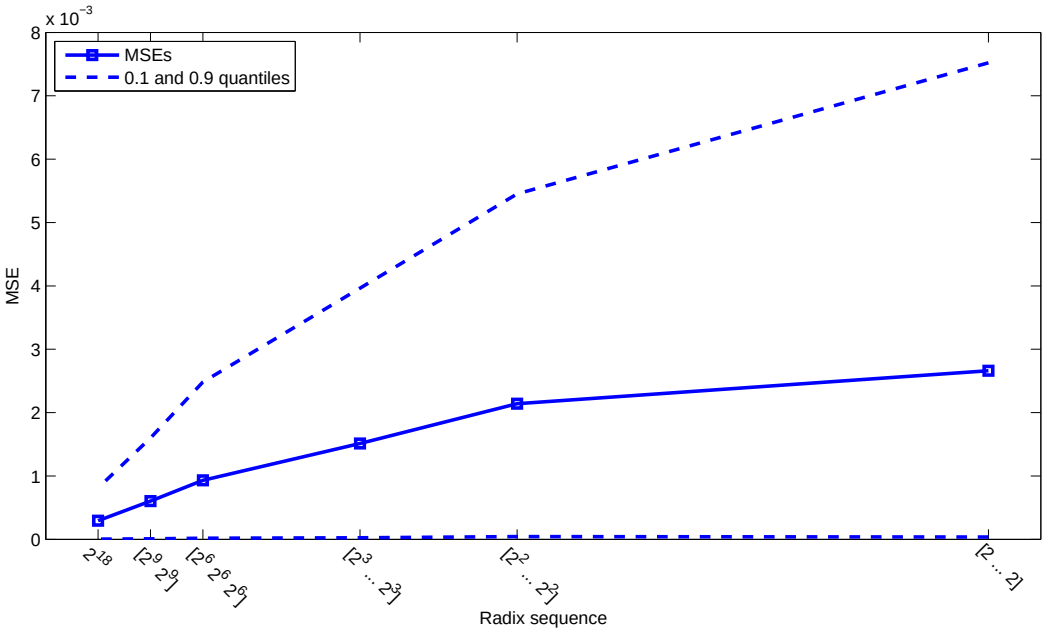


Figure 4.3. Fixed radix, variable depth case, dashed lines show the 0.1 and 0.9 quantiles of squared errors.

4.1.4. Comparison of Butterfly and Multinomial Resampling on GPU

In this section, our aim is to compare the perfomance of parallel implementations of multinomial resampling and butterfly resampling on a GPU in terms of speed and MSE. More specifically, we wish to illustrate that the butterfly resampling scheme can provide a significant speed-up over multinomial resampling, while keeping the estimation error, as measured by MSE at a competitive level. Our assumption is

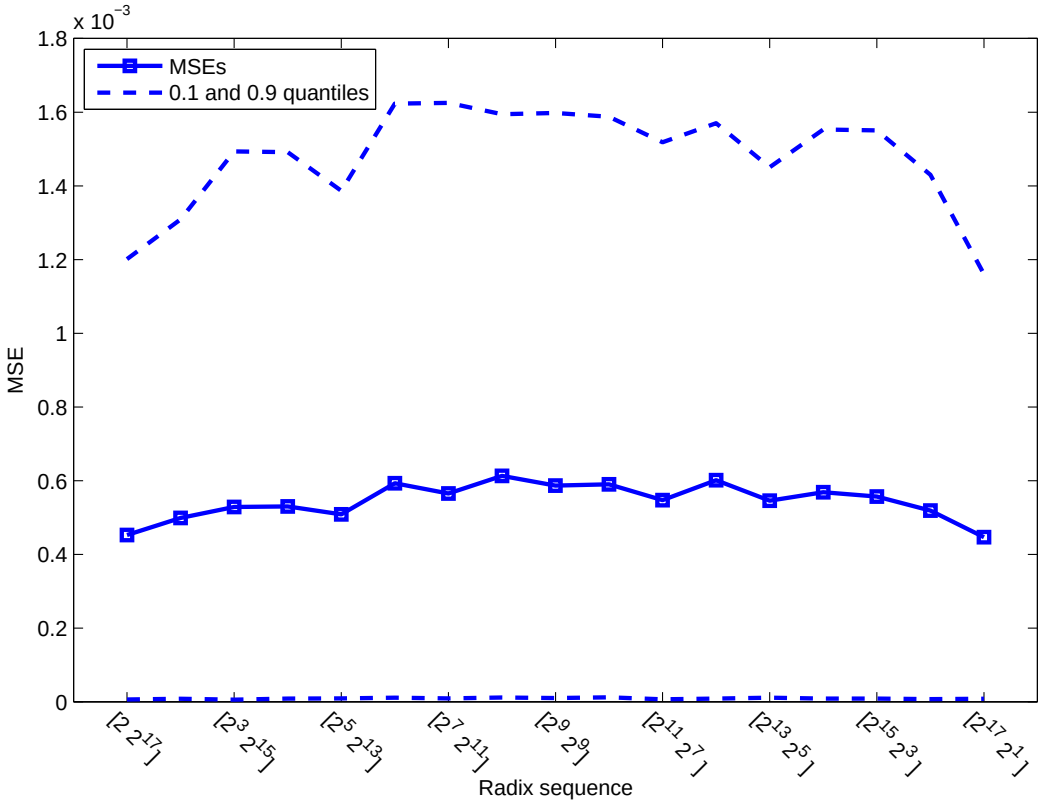


Figure 4.4. Mixed radix, fixed depth case, dashed lines show the 0.1 and 0.9 quantiles of squared errors.

that local interaction structure of butterfly resampling, as opposed to full interaction structure of standard multinomial resampling, fits better into constraints imposed by CUDA programming model.

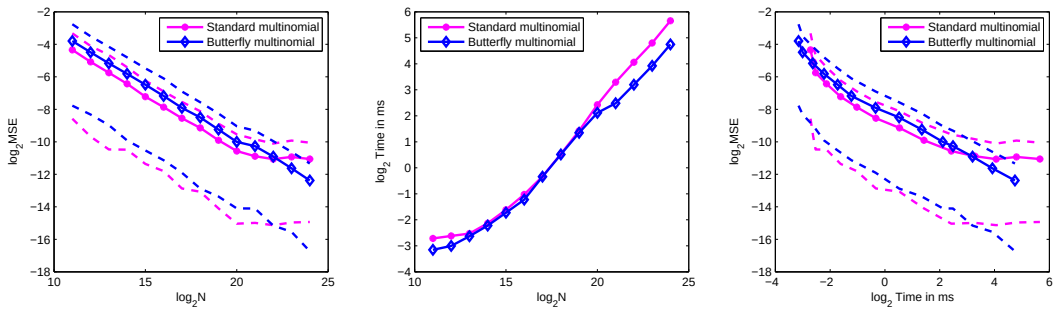
We conduct two experiments to estimate and compare MSEs and speeds of standard multinomial and butterfly resampling schemes, two experiments differ by the choice of particle-weight generation processes. For a given initial distribution π_0, potential function g and particle number N, we generate $T = 1000$ particle sets $(x_{0,t})_{t=1}^{T}$ independently, where particles within a particle set are i.i.d from the distribution π_0. We perform the resampling algorithms with inputs $(x_{0,t}, w_{0,t} = (w_{0,t}^i = g(x_{0,t}^i))_{i=1}^{N})_{t=1}^{T}$ to produce T ancestor vectors $(a_t = (a_t^i)_{i=1}^{N})_{t=1}^{T}$ such that $\xi_{m,t}^i = x_{0,t}^{a_t^i}$. Finally, we compute $\widehat{\mathrm{LMSE}}_{\varphi}^{N}$ as in Equation 4.6 and plot it against particle number and running time of resampling algorithm. We repeat this procedure for number of particles $N = 2^{11}, 2^{12}, \ldots, 2^{22}$.

Since butterfly resampling algorithm can be parameterized with radix sequence, we use the observations we made in Section 4.1.3 for guidance in the choice of radix sequence: it is preferable to allow for maximum interaction that can be implemented efficiently in the parallel platform, resulting in as shallow radix sequences as possible. In our CUDA implementations of butterfly resampling, interaction between particles are characterized by the size of thread blocks. Each particle is mapped to a thread and each set of interacting particles mapped to a thread block, so the maximum radix in our CUDA implementation is bounded by the maximum number of threads per block, r_{max}, which is 1024 for the latest CUDA specification. Combining this and above considerations, in our experiments we use radix sequences of depth $m_N = \lceil \log_{r_{max}} N \rceil$. We also note that this choice also reduces the running time, as deeper radix sequences will increase the number of parallel stages, effectively increasing the running time.

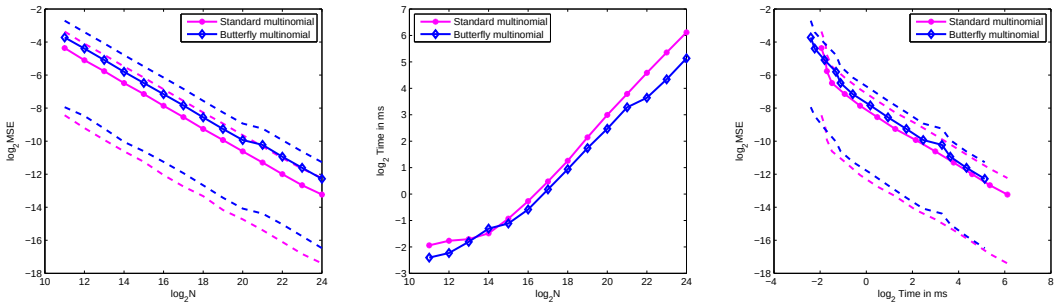
Once the depth of the radix sequence, m_N, is fixed, we run butterfly resampling algorithm for all possible radix sequences with depth m_N and report the results for the fastest one. For example, when $N = 2^{12}$, we have $m_N = 2$, so we run our algorithm with radix sequences $(2^2, 2^{10}), (2^3, 2^9), \ldots, (2^{10}, 2^2)$. Empirical results indicate that butterfly algorithm with more uniform radix sequences i.e. radix sequences with radices close together, tend to run faster. While there are a lot of factors having effect on speed of GPU implementation of butterfly resampling, ranging from hardware specifications of GPU, like maximum number of thread blocks per streaming multiprocessor or maximum shared memory size per block, to particular weight distribution, we argue that large radices in nonuniform radix sequences effectively decrease memory bandwidth because of strided memory accesses with large strides. In Section 4.1.3, we observed that, for a given number of particles N, Monte Carlo error measured by MSE does not depend very much on the choice of radix sequence when the depth is fixed. Combining our empirical results on MSE and speed, we conclude that it is preferable to use more uniform radix sequences in practice.

In the first experiment, we generate our particle-weight sets as follows : particles x_t^i are independently sampled from a uniform distribution on $[-\sigma, \sigma]$ and the potential function is unimodal $g(x) = \exp\{-x^2/2\sigma^2\}$, where $\sigma = 10$. In the second and

third columns of the Figure 4.5. we plot runtimes and MSE versus particle numbers, respectively, while in the first column of the same figure we plot MSE versus runtime.



(a) Single precision.

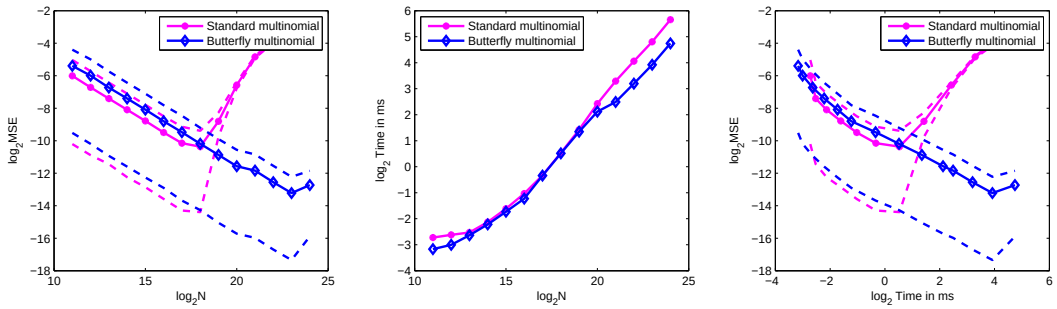


(b) Double precision.

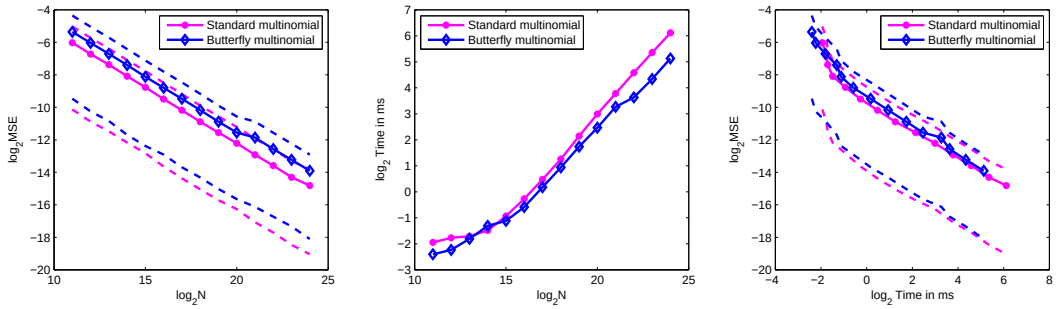
Figure 4.5. Comparison of standard and butterfly multinomial resampling, potential function $g(x) = \exp\{-x^2/2\sigma^2\}$.

In the second experiment, we generate our particle-weight sets as follows : particles x^i are independently sampled from a Poisson distribution with intensity parameter λ and the potential function is the probability mass function of the same distribution $g(x) = \lambda^x/x!$, where $\lambda = 100$. As in the first experiment, in the second and third columns of Figure 4.6. we plot runtimes and MSE versus particle numbers, respectively, and in the first column of the same figure we plot MSE versus runtime.

As we can see in figures Figure 4.5. and Figure 4.6., butterfly resampling scheme is in general faster than standard multinomial resampling, it provides of to two times speed-up, while it has slightly worse Monte Carlo error. If we look at the Monte Carlo error produced per computation time, standard multinomial resampling is slightly better. We see that for large numbers of particles, in single precision arithmetic context, standard multinomial resampling becomes numerically unstable while butterfly resam-



(a) Single precision.



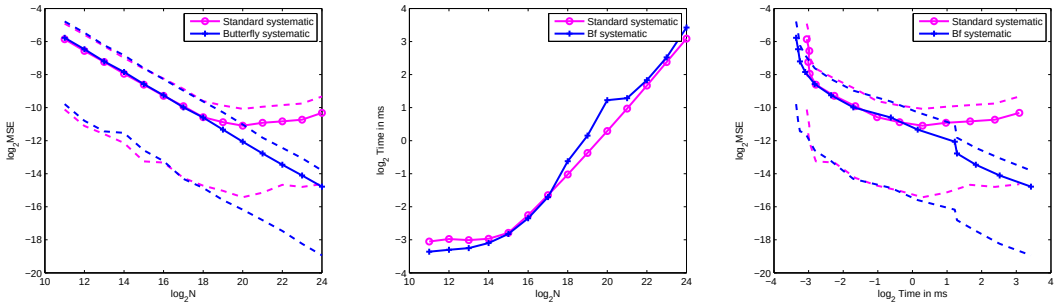
(b) Double precision.

Figure 4.6. Comparison of standard and butterfly multinomial resampling, potential function $g(x) = \lambda^x/x!$.

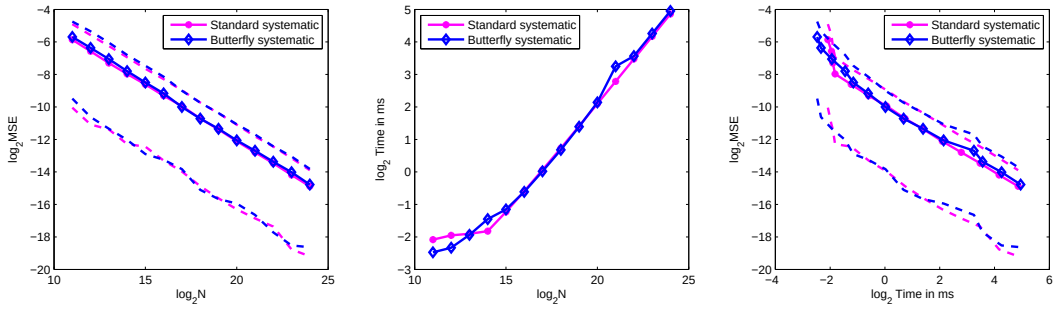
pling stays stable. We do not see this behaviour in double precision arithmetic context, however, in GPUs single precision arithmetic is substantially faster and it is preferable to carry out computations in single precision when we do not need extremely high numerical precisions.

We repeat these experiments within the same setting for standard and butterfly systematic resampling algorithms: particle generation processes, experimental setups regarding particle numbers, number of repeats and radix sequences are the same as above two experiments. In figures Figure 4.7.a and Figure 4.7.b we plot runtimes and MSE results for single precision and double precision implementations of standard and butterfly systematic resampling algorithms. In figures Figure 4.8.a and Figure 4.8.b we plot runtimes and MSE results for single precision and double precision implementations of standard and butterfly systematic resampling algorithms. In parallel with above experiments, we see that butterfly scheme does not provide an improvement in terms of MSE per time and standard systematic resampling algorithm becomes

numerically unstable as we increase the number of particles N in single precision implementations. On the other hand, we see that MSE difference between two algorithms is less than that of multinomial scheme and butterfly systematic resampling does not provide a visible speed-up over standard systematic resampling algorithm.



(a) Single precision.

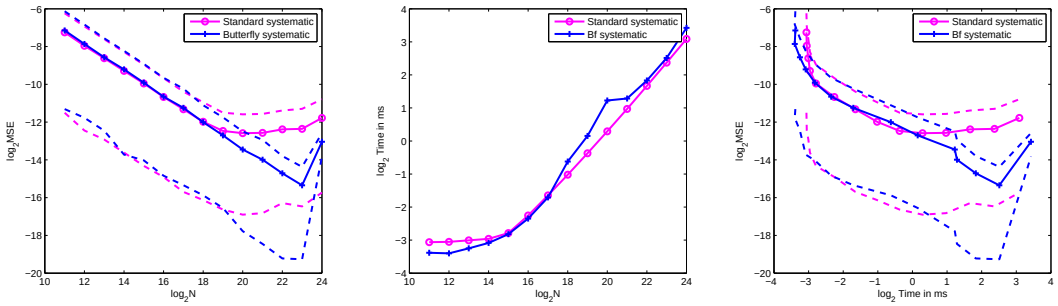


(b) Double precision.

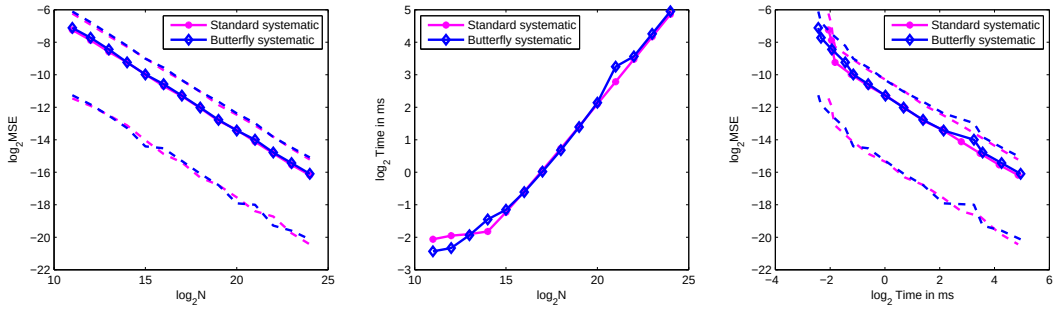
Figure 4.7. Comparison of standard and butterfly systematic resampling, potential function $g(x) = \exp\{-x^2/2\sigma^2\}$.

4.2. Experiments with Particle Filters

In this section we compare butterfly multinomial resampling with standard multinomial resampling in GPU implementations of particle filter. We compare performances of two resampling schemes in terms of Monte Carlo error and computation speed by considering two particle filtering scenarios : bootstrap particle filter (Section 4.2.2) and adaptive resampling particle filter (Section 4.2.3). Before moving on to experiments we describe the setting in which we evaluate our algorithms.



(a) Single precision.



(b) Double precision.

Figure 4.8. Comparison of standard and butterfly systematic resampling, potential function $g(x) = \lambda^x/x!$.

4.2.1. General Setting

As before, we consider the butterfly resampling algorithm parameterized by a radix sequence $(r_1, \ldots, r_m)$ such that $\prod_{k=1}^{m} r_k = N$ and write $\xi_{0,t} = (\xi_{0,t}^i)_{i=1}^N$ for the input particles and $w_{0,t} = (w_{0,t}^i)_{i=1}^N$ for the input weights of first stage of butterfly resampling, $\xi_{k,t} = (\xi_{k,t}^i)_{i=1}^N$ for the random output particles and $w_{k,t} = (w_{k,t}^i)_{i=1}^N$ for the output weights of the kth stage of the butterfly resampling algorithm at time step t of particle filter. Then, the output empirical measures of the bootstrap particle filter are

$$\pi_t^N = \frac{1}{N}\sum_{i=1}^{N} \delta_{\xi_{0,t}^i}, \quad \hat{\pi}_t^N = \frac{1}{N}\sum_{i=1}^{N} \delta_{\xi_{m,t}^i}, \quad \text{for } t \geq 1.$$

In the case of adaptive resampling particle filter, we have another sequence of random variables, $\{K_t\}_{t\geq 1}$ with support $\{0, 1, \ldots, m\}$, the number of butterfly resampling stages performed at time step t of particle filter, then output empirical measures of

adaptive resampling particle filter are

$$\pi_t^N = \frac{\sum_{i=1}^N w_{0,t-1}^i \delta_{\xi_{0,t}^i}}{\sum_{i=1}^N w_{0,t-1}^i}, \quad \hat{\pi}_t^N = \frac{\sum_{i=1}^N w_{K_t,t}^i \delta_{\xi_{K_t,t}^i}}{\sum_{i=1}^N w_{K_t,t}^i}, \quad \text{for } t \geq 1.$$

Note that, bootstrap particle filter is a special case of adaptive resampling particle filter, in which we have $\mathbb{P}(K_t = m) = 1$ for all $t \geq 1$.

To quantify the Monte Carlo error associated to particle filter, we use MSE averaged over time steps defined as follows :

$$\text{AMSE}_\varphi^{N,T} = \mathbb{E}\left[\frac{1}{T}\sum_{t=1}^T \left(\hat{\pi}_t^N(\varphi) - \hat{\pi}_t(\varphi)\right)^2\right],$$

where the expectation is taken over all realizations of the particle filter.

We estimate $\text{AMSE}_\varphi^{N,T}$ for the standard multinomial resampling and the butterfly resampling by numerical simulation experiments. For a given number of particles, N, and a test function, φ, we run the particle filter at hand M times to get the empirical measures $\hat{\pi}_{t,j}^N$ for $t = 1, \ldots, T$ and $j = 1, \ldots, M$. Then estimated $\text{AMSE}_\varphi^{N,T}$ is calculated as follows :

$$\widehat{\text{AMSE}}_\varphi^{N,T,M} = \frac{1}{M}\sum_{j=1}^M \left(\frac{1}{T}\sum_{t=1}^T \left(\hat{\pi}_{t,j}^N(\varphi) - \hat{\pi}_t(\varphi)\right)^2\right).$$

In the experiments of next two subsections, we use a simple model with a multimodal filtering distribution given as follows :

$$x_{t+1} \sim f(x_t, .) = \mathcal{N}(x_t, \sigma_1) \tag{4.7}$$

$$g_t(x) = 0.5\exp\{(x - 0.5t)^2/2\sigma_2^2\} + 0.5\exp\{(x + 0.5t)^2/2\sigma_2^2\}, \tag{4.8}$$

where the exact values of expectations under filtering distribution can be computed easily. Under this model, we have $\hat{\pi}_t(\varphi) = 0$ for $\varphi(x) = x$. All particle filter imple-

mentations use the transition density of this model, f, as the proposal density of the particle filter.

For the parameterization of butterfly resampling algorithm, we follow the guidelines we obtained from the empirical results of the Section 4.1.3. We use as uniform radix sequences as possible with smallest possible depths, $m_N = \log_{r_{max}} N$ where $r_{max} = 1024$, with nonincreasing radices. For example, when $N = 2^{22}$, we use the radix sequence $(2^8, 2^7, 2^7)$.

For a given particle filter algorithm and number of particles N, we run the particle filter $M = 1000$ times and compute $\widehat{\mathrm{AMSE}}_{\varphi}^{N,T,M}$, for $T = 100$ and $\varphi(x) = x$. We repeat this experiment for particle numbers $N = 2^{11}, \ldots, 2^{22}$ and plot the MSE and speed results.

4.2.2. Bootstrap Particle Filter

In this subsection, we want to show that the butterfly resampling scheme can be a competent alternative to multinomial resampling, in the context of bootstrap particle filter. For this, we compare two different implementations of the bootstrap particle filter, one that uses standard multinomial resampling given in Figure 2.2. and another one that uses full-depth butterfly resampling given in Figure 3.6. We conduct our experiments within the setting desribed above. Results of these experiments are plotted in Figure 4.9.

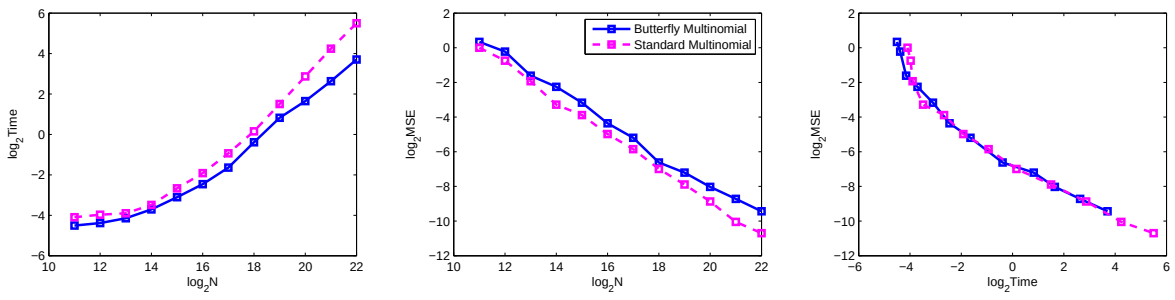


Figure 4.9. Performance comparison of bootstrap particle filters.

As in the case of single step of resampling algorithms, we see that, particle filter that employs butterfly resampling algorithm have better speed, it provides upto four

times speed-up, but worse Monte Carlo error. If we compare the Monte Carlo error per computation time, again we see that standard multinomial resampling has a slight advantage. For large number of particles we see the effect of numerical unstability of standard multinomial resampling in the Monte Carlo error.

4.2.3. Adaptive Resampling Particle Filter

In this subsection, we want to show the performance advantage of using butterfly resampling in an adaptive resampling particle filter context. For this, we compare two different implementations of adaptive resampling algorithm, one that uses adaptive butterfly resampling given in Figure 3.7. and one that uses adaptive multinomial resampling which is a special case of Figure 3.7. with $m = 1$ and $r = (N)$. We conduct our experiments within the setting desribed above except that we set the ESS threshold to $\tau = 0.6$ and plot their results in Figure 4.10.

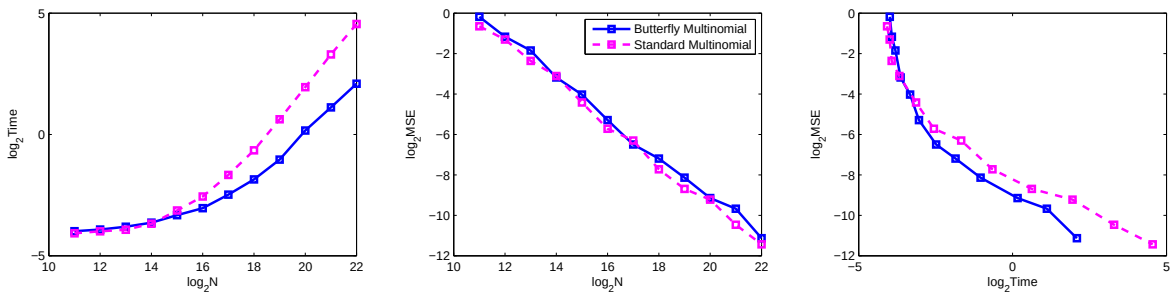


Figure 4.10. Performance comparison of adaptive resampling particle filters.

In Figure 4.10. we see that the speed-up of butterfly resampling over standard multinomial resampling has increased, it is upto six times faster than standard multinomial implementation, and difference between Monte Carlo error is decreased, as a result, performance of butterfly resampling is better than standard multinomial resampling in terms of Monte Carlo per computation time. We can say that by adaptively choosing the level of interaction we can exploit the incremental structure of butterfly resampling mechanism.

In our last experiment, we compare the level of interactions of butterfly resampling and standard multinomial resampling algorithms in the adaptive resampling par-

ticle filter context. For this, we use the above generative model 4.7 with number of particles $N = 1024$ and ESS threshold $\tau = 0.6$. In the butterfly resampling algorithm, we set the radix sequence to be $r = (2, 2, \ldots, 2)$, which has depth of 10, since $1024 = 2^{10}$. The level of interaction is measured by base 2 logarithm of the size of biggest interacting particle block formed by the algorithm. For standard multinomial resampling algorithm this number is either 0, when resampling is not performed, or 10, when resampling is performed. For butterfly resampling, if algorithm performs k stages the level of interaction is k. We plot the ESS and level of interaction at every time step of particle filters in Figure 4.11.

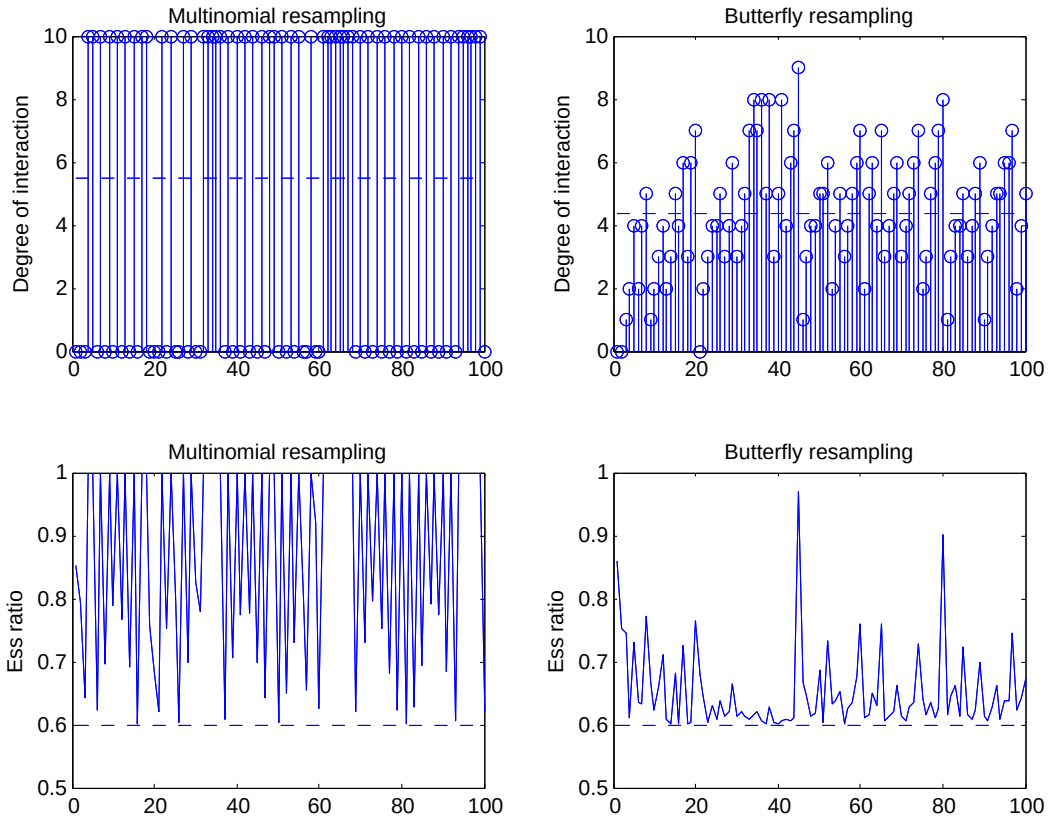


Figure 4.11. ESS and interaction level comparisons of resampling algorithms.

In Figure 4.11. we see that the average level of interaction of butterfly resampling particle filter is less than that of standard multinomial resampling. With this observation in mind, we can say that adaptive butterfly resampling particle filter can adapt its parsimony better in terms of computational resources in expense of keeping the ESS at a lower level. This kind of parsimony explains the performance improvement of butterfly resampling algorithm over multinomial resampling in adaptive resampling

particle filter compared to bootstrap particle filter.

4.3. Comparison of Resampling Algorithms in a Practical Application

As a final experiment, we compare performance of butterfly resampling scheme to classical resampling schemes in a more practical context. In this experiment, we use particle filter approximations of the smoothing distribution $p_n(dx_{0:n}) = \mathbb{P}(X_{0:n} \in dx_{0:n} | \{Y_{0:n} = y_{0:n}\})$ to perform maximum likelihood (ML) estimation via Expectation-Maximization (EM) algorithm [62]. EM algorithm is an iterative process that uses an approximation of ML estimator to find a lower bound function to likelihood function and optimizes this lower bound to get a better approximation of ML estimator.

Assume that we have a state space model on a family of probability spaces $(\Omega, \mathcal{F}, \mathbb{P}_\theta)$ parameterized by a set of parameters $\theta = (\theta^1, \ldots, \theta^d) \in \Theta$ with typically $\Theta = \mathbb{R}^d$ such that

$$\begin{aligned} X_0 &\sim \mu_\theta, \\ X_n \mid \{X_{n-1} = x_{n-1}\} &\sim f_\theta(x_{n-1}, \cdot), \\ Y_n \mid \{X_n \ = \ x_n\} &\sim g_\theta(x_n, \cdot), \end{aligned} \tag{4.9}$$

where μ_θ, $f_\theta(x, \cdot)$ and $g_\theta(x, \cdot)$ are all dominated by reference measures on appropriate spaces for all $x \in \mathsf{X}$, $\theta \in \Theta$ and Radon-Nikodym derivatives are also denoted respectively by μ_θ, f_θ and g_θ with a slight abuse of notation. For a given observation sequence $y_{0:n}$, we want to find the value of θ that maximizes the log-likelihood

$$\ell(\theta) = \log(\mathbb{P}_\theta(Y_{0:n} \in dy_{0:n})) = \log\left(\int \mu_\theta(x_0) \prod_{i=1}^{n} f_\theta(x_{i-1}, x_i) \prod_{i=0}^{n} g_\theta(x_i, y_i) dx_{0:n}\right).$$

EM algorithm starts with an initial guess θ' of the parameter and recursively improves the estimate by maximizing a lower bound $\mathcal{Q}(\theta, \theta')$ of the integral on the right hand side of the above equation with respect to θ which is in the form of the conditional

expectation with respect to $\mathbb{P}_{\theta'}$:

$$\begin{aligned}\mathcal{Q}(\theta,\theta') =& \mathbb{E}_{\theta'}\left[\sum_{i=0}^{n-1}\log f_\theta(x_i,x_{i+1})\middle|\{Y_{0:n}=y_{0:n}\}\right]+\mathbb{E}_{\theta'}\left[\sum_{i=0}^{n}\log g_\theta(x_i,y_i)\middle|\{Y_{0:n}=y_{0:n}\}\right]\\ &+\mathbb{E}_{\theta'}[\log\mu_\theta(x_0)|\{Y_{0:n}=y_{0:n}\}]\end{aligned}$$

Two steps of EM algorithm at the iteration t can be written as:

(i) *Expectation :* Compute the expectation $\mathcal{Q}(\theta,\theta_{t-1})$ where θ_{t-1} is the output of the previous iteration as a function of θ,

(ii) *Maximization :* Maximize $\mathcal{Q}(\theta,\theta_{t-1})$ over θ, set $\theta_t=\text{argmax}_\theta\,\mathcal{Q}(\theta,\theta_{t-1})$ to get a new estimate of parameter.

Repeating this procedure we obtain a sequence $(\theta_t)_{t\geq 0}$ that converges to a stationary point of log-likelihood $\ell(\theta)$. However, conditional expectations in this expression are intractable for a majority of practical models and we have to use approximations instead of exact values of these expectations. This modification to EM algorithm will produce converging sequences for certain families of probability models [63]. Particle filters come into play at this point, note that conditional expectation in the above equation are expectations under smoothing distribution $p_{\theta,n}(dx_{0:n})=\mathbb{P}_\theta(X_{0:n}\in dx_{0:n}|\{Y_{0:n}=y_{0:n}\})$ instead of which we can use the particle approximation $p^N_{\theta,n}=\sum_{i=1}^N\frac{w_n^i}{\sum_j w_n^j}\delta_{\hat{\xi}^i_{0:n}}(\cdot)$. The instance EM algorithm that uses particle filter approximations of smoothing distributions is called Sequential Monte Carlo EM (SMCEM) algorithm [56].

In our implementations of SMCEM algorithm, we exploit the additive form of the expectations in $\mathcal{Q}(\theta,\theta')$ function. Let $(t_n(x_{0:n}))_{n\geq 1}$ be sequence of additive functionals in the form of

$$t_n(x_{0:n})=\sum_{i=1}^n s_i(x_{i-1},x_i)=t_{n-1}(x_{0:n-1})+s_n(x_{n-1},x_n),n\geq 1,$$

where $(s_i)_{i\geq 1}$ are measurable functions. We want to compute

$$p_n(t_n(x_{0:n})) = \sum_{i=1}^{N} \frac{w_n^i}{\sum_j w_n^j} t_n(\hat{\xi}_{0:n}^i) = \sum_{i=1}^{N} \frac{w_n^i}{\sum_j w_n^j} \{t_{n-1}(\hat{\xi}_{0:n-1}^i) + s_n(\hat{\xi}_{n-1}, \hat{\xi}_n)\}.$$

This equation suggests that, instead of keeping the whole particle paths, we can keep the evaluated values of t_n for $n \geq 1$ at each time step of the particle filter. Although repeated resampling mechanism cause the sample impoverishment problem, the effect that initial segments of particle trajectories collapse into one path, we use this simple approach for the ease of implementation and since our objective is not the parameter estimation but the comparison of resampling algorithm in terms of speed and Monte Carlo variance.

4.3.1. Experiment Setup

A particularly interesting set of models in practice consists of the distributions coming from the exponential family, i.e. the models with joint probability density of $X_{0:n}$ and $Y_{0:n}$ given the parameters is of ther form $\exp(\langle \psi(\theta), S_n(x_{0:n}) \rangle - c(\theta))h(x_{0:n})$ where ψ is a $\mathbb{R}^d$ valued function of parameters, S_n are $\mathbb{R}^d$ valued functions of hidden paths for some $d \geq 1$ and $\langle \cdot, \cdot \rangle$ denotes the usual dot product in Euclidean space. In such a model, our objective function of EM algorithm takes the simple form:

$$\mathcal{Q}(\theta, \theta') = \langle \psi(\theta), p_{\theta',n}(S_n) \rangle - c(\theta)$$

for which we use the particle filter estimates $\widehat{S}_n = p_{\theta',n}^N(S_n)$ instead of the exact values of expectations. In our last experiment, we follow the examples of applications of SMC methods to ML estimation problems presented in [56] and apply Sequential Monte Carlo EM (SMCEM) algorithm with different resampling strategies to stochastic volatility model [64] which has the above desribed form. The model description of

stochastic volatility model is as follows:

$$X_0 \sim \mathcal{N}(0, \sigma^2),$$
$$X_n \sim \mathcal{N}(\alpha X_{n-1}, \sigma^2), n \geq 1,$$
$$Y_n \sim \mathcal{N}(0, \beta^2 \exp(X_n)), n \geq 0.$$

For stochastic volatility model we have $\psi(\theta) = (-\alpha/(2\sigma^2), -1/(2\sigma^2), \alpha/\sigma^2, -1/(2\beta^2))$, $c(\theta) = (n+1)\log(\beta^2)/2 + (n+1)\log(\sigma^2)/2$ and $S_n = (S_{n,1}, S_{n,2}, S_{n,3}, S_{n,4})$ such that $S_{n,1} = \sum_{k=0}^{n-1} x_k^2$, $S_{n,2} = \sum_{k=0}^{n} x_k^2$, $S_{n,3} = \sum_{k=0}^{n-1} x_k x_{k+1}$ and $S_{n,4} = \sum_{k=0}^{n} y_k^2 \exp(-x_k)$. Plugging particle filter estimates $\widehat{S}_n = (\widehat{S}_{n,1}, \widehat{S}_{n,2}, \widehat{S}_{n,3}, \widehat{S}_{n,4})$ of these components into $\mathcal{Q}(\theta, \theta')$, taking the partial derivatives with respect α, β and σ, we obtain the EM update rule:

$$\alpha = \frac{\widehat{S}_{n,3}}{\widehat{S}_{n,1}}, \quad \beta^2 = \frac{\widehat{S}_{n,4}}{n+1}, \quad \sigma^2 = \frac{1}{n+1}\left(\alpha^2 \widehat{S}_{n,1} - 2\alpha \widehat{S}_{n,3} + \widehat{S}_{n,2}\right) \tag{4.10}$$

In our experiments, we will simulate M observation sequences $(y_{0:n}^m)_{m=1}^M$ of length n, with parameter values $\alpha^* = 0.975$, $\beta^* = 0.63$ and $\sigma^* = 0.16$. For each of the M observation sequences, we will perform SMCEM algorithm and obtain a total of M parameter estimates $(\theta^m)_{m=1}^M$ and compute MSE of each parameter. In SMCEM algorithm, we perform T EM iterations and within each EM iteration we run a particle filter with N particles. We perform this procedure for bootstrap and adaptive resampling particle filters with standard and butterfly multinomial resampling mechanisms.

4.3.2. Experiment Results

In this subsection we present our results for the experiments we conducted within the setup described in previous subsection. We repeat the experiment for particle numbers $N = 2^{11}, 2^{13}, 2^{15}, 2^{17}, 2^{19}$ and 2^{21}. For each particle number N, we generate $M = 10$ observation sequences of length $n = 100$ and perform SMCEM algorithm with $T = 100$ EM iterations using for two different particle filter algorithms: one that uses

multinomial resampling and another one that uses butterfly resampling. In our particle filter implementations we use the model transition density as our proposal density. To simulate a situation with hard communication constraints, we set the maximum radix sequence to 256 and we use an unusual strategy for resampling.

In the particle filter implementation with multinomial resampling we perform resampling at every $\lceil \log_{256} N \rceil$th time step. In the particle filter implementaion with butterfly resampling, we perform one stage of resampling at every time step and in order to avoid degeneracy we permute particles at each time step, since otherwise we would have run small independent particle filters in parallel. For the choice of radix sequence of the butterfly resampling algorithms we follow the guidelines we obtained in Section 4.1.3 but since we perform only one stage we use only the first radix in the sequence. We fix the β parameter to $\beta^* = 0.63$ and observe α and σ parameters.

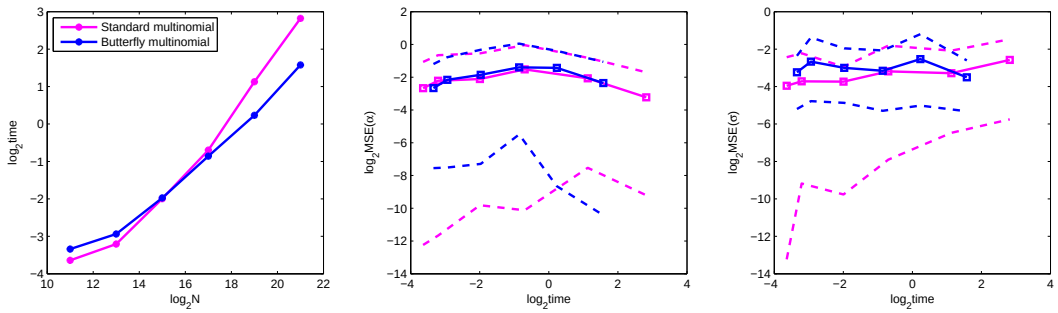


Figure 4.12. Performance comparison of resampling algorithms in SMCEM algorithm.

We plot the mean computation time per EM iteration in the leftmost plot of Figure 4.12., MSE of the parameter estimates per mean computation time for α parameter in the middle plot in Figure 4.12. and MSE of the parameter estimates per mean computation time for σ parameter in the rightmost plot in Figure 4.12. We see that butterfly resampling algorithm becomes faster as we increase the number of particles and achieves about four times speed-up over multinomial resampling for $N = 2^{21}$ particles. Although the variance of the estimates of $\mathcal{Q}(\theta, \theta')$ made by particle filter employing butterfly resampling is more than the one with multinomial resampling, we see that quality of parameter estimations does not significantly differ for two algorithms as the estimation error of ML method depends on the number of observations which is fixed for all SMCEM runs in this experiment.

5. CONCLUSIONS AND FUTURE WORK

5.1. Conclusions

In this thesis, we investigated the implementation of particle filters on GPUs and presented the recently proposed augmented resampling framework and butterfly resampling scheme as its special case for the resampling part of the particle filters in addition to classical resampling algorithms originally propsed in the context of serial computation. First, we described the problem of filtering in the Bayesian setting and reviewed the particle filtering algorithms in the context of hidden Markov Models. Particle filtering algorithms are based on the importance sampling scheme, which is used when we cannot directly sample from a distribution. In Section 1.2 we reviewed the Sequential Importance Sampling (SIS) and Sequential Importance Resampling (SIR) algorithms as the prototypical instances of the modern particle filtering methods and gave the generic form of the particle filtering algorithm that consists of three main parts: sampling of particles, computation of weights and resampling. While sampling and weight computation parts are naturally parallel, resampling part needs special attention when resampling and in this thesis, we focused on the implementation of particle filters on parallel platforms via parallelization of resampling part.

In Section 1.3 we described a generic form of resampling procedure that allows us the identify the interplay between the underlying stochastic process of resampling algorithm and suitability of algorithm to parallelization on a given platform. The interactions between random variables involved in the resampling algorithms are characterized by the conditional independence structure imposed by the algorithm which can be represented by a graph we call the interaction structure. The key to build resampling mechanism that can run efficiently on some parallel platform is to map this interaction structure to the network structure on the computational units formed by the modes of communication between them.

In Section 1.4 we reviewed the GPU programming model of CUDA platform

developed and maintained by NVIDIA. The massively parallel architecture of CUDA platform is organized into threads, which are basic units of the computational components, and thread blocks, the group of threads that can work cooperatively on a task. This hierarchical structure and the constuctions of global memory and shared memory constitutes a communication network: while threads within a block can communicate with each other via fast shared memory and barrier synchornizations, threads from different blocks can communicate only through read and write operations to global memory which is much slower compared to shared memory. The hardware implementation of massively parallel architecture complicates the design of efficient programs on GPUs. Poor latency of the global memory demands the utilization of coalesced memory accesses and to avoid thread divergence programmer has to pay special attention to the mapping between the indices of the tasks and the threads.

In Chapter 2 we investigated the parallel implementations of classical resampling methods: multinomial resampling in Section 2.1, stratified resampling in Section 2.2 and systematic resampling in Section 2.3. We identify the factors that facilitates or hinders their parallelization by observing their conditional independence structures. The dependence of output particles on all input particles induces a full interaction structure which necessitates heavy communication between the components of the parallel computation platform. Although this conditional independence structure hinders the parallelization of multinomial resampling, it gives rise to $\sqrt{N}$ scaling of the asymptotic fluctuations of estimation errors for the particle filter that employs the standard multinomial resampling algorithm. We also note that, although stratified and systematic resampling algorithms does not put any restrictions on the interaction between input and output particles, conditioned on the input weights and the ordering of the input particles, these resamplings schemes impose the dependence of output particles only on a fraction of input particles because of the stratification.

Our analysis suggest that the sampling part of the parallel implementation of standard multinomial resampling algorithm has a $\mathcal{O}(t_G \log_2 N + t_K)$ time cost where t_G, t_K are device specific constants and N is the number of particles. The sampling parts of the parallel implementations of stratified and systematic resampling algorithm have

the same asymptotic time cost of $\mathcal{O}(ct_G+t_K)$ on average, it does not depend on number of particle N. Here, c is a constant depending on the probability model governing the dynamics of the joint process (X, Y). Stratified and systematic resampling methods are superior to multinomial resampling algorithm both in terms of computation cost and Monte Carlo error associated to the resampling mechanism, yet the theory of multinomial resampling is better understood and it serves as a reference point for the convergence results of the butterfly resampling algorithms.

In addition to GPU implementations of classical resampling algorithms, we presented two recently proposed resampling algorithms in Section 2.4. Metropolis resampling and rejection resampling algorithms are proposed specifically for the implementation on GPUs. One of their key properties is that they do not require the computation of prefix sum which causes a high degree of communication between particles. On the other hand, like classical resampling algorithms, these methods does not put any constraint on the dependence of output particles on the input particles. Altough removal of the prefix sum computation reduces the computation cost substantially, this full interaction structure can cause a heavy memory traffic via random global memory accesses. Apart from this their implementation on GPUs pose additional challenges as their parameterization is not straightforward and bias of the Metropolis resampling scheme can put the consistency and convergence properties of the particle filter into jeopardy.

We reviewed the idea of resampling under constrained interaction structure in Chapter 3 and presented the resampling method of αSMC as a foundation to the augmented resampling framework. Augmented resampling framework presented in Section 3.1 provides flexibility in designing resampling algorithms intended to run on a specific parallel architecture. We see that the hardware and software architectures of a parallel computing platforms impose a structure on the patterns of communication between computational units and augmented resampling framework allow us to tailor the interaction structure of particle involved in the resampling procedure to these structured communication patterns. We state the unbiasedness property and moment bounds for a certain class of augmented resampling framework in which we consider

m-fold factorizations of the matrix $\mathbf{1}_N$, an approach that leads to generalizations of classical resampling algorithms.

As an instance of augmented resampling framework, we present the butterfly resampling algorithm in Section 3.2, which can be especially suited to the communication structure of threads in a GPU imposed by thread blocks, barrier synchronization and shared memory structures. The main feature of the butterfly resampling is that it allows us to divide the full interaction into stages of constrained interactions. It divides particles into smaller blocks and perform resampling within these blocks, then proceeds by applying this process by partitioning the particles into a different set of blocks. The resampling operation within blocks can be carried out via classical resampling algorithms. We compared butterfly resampling algorithms to their classical counterparts, which can be derived as special cases of butterfly resampling algorithm, in terms of Monte Carlo error and computational cost. Butterfly multinomial resampling algorithm given in Figure 3.6. requires $\mathcal{O}(t_S \log N + t_K log_{r_{\max}} N)$ time and butterfly stratified and systematic resampling algorithms require $\mathcal{O}(c' t_S + t_K \log_{r_{\max}} N)$ time on GPUs, where t_S is the average time to access a shared memory location, .

Theorem 3.4 states that when we keep the level of interaction at a fixed level while increasing the number of particles, we get exotic scaling $\sqrt{N/\log N}$ of asymptotic errors as opposed to $\sqrt{N}$ scaling of multinomial resampling. This is the cost of more parallelizable interaction structure of butterfly resampling. To get the error scaling of multinomial resampling, we can limit the number of stages in butterfly resampling. Theorem 3.5 states that when we fix the number of stages to two and keep the level of interaction in the first stage at a fixed level, we get $\sqrt{N}$ scaling of multinomial resampling but with a greater asymptotic variance. Although these two scenarios are very specific cases of butterfly resampling scheme, they give a taste of what statistical properties butterfly resampling can have. These kinds of results for the most general case of butterfly resampling is much harder and requires a deep combinatorial analysis of the butterfly graphs.

In Section 3.3 we presented a practical extension of butterfly resampling algo-

rithm, adaptive butterfly resampling algorithm. Its main feature is that by choosing the degree of interaction adaptively it can exploit the incremental nature of the butterfly resampling algorithm. While ESS computation incurs an additional computation cost, we can gain from doing away with some stages of resampling algorithm. In our experiments with particle filters, we see that this is indeed the case and in an adaptive resampling particle filter scenario speed-up of butterfly multinomial resampling algorithm over standard multinomial resampling increases and butterfly resampling algorithm becomes better in terms of Monte Carlo error per computation time.

In Chapter 4, we presented the experiments we conducted to verify the theoretical results on butterfly resampling algorithms by empirical results, to obtain practical guidelines for the parameterization of butterfly resampling and to compare classical resampling algorithms to butterfly resampling algorithms in terms of speed and Monte Carlo error. To do this first we describe the setting in which we conduct and evaluate our experiments. In Section 4.1, we present our results concerning single step of resampling isolated from a particle filter context. We verify the result of Theorem 3.4 and obtain empirical results that give us practical guidelines for the implementation of butterfly resampling algorithm. We see that although butterfly multiomial resampling algorithm can provide a substantial speed-up, if we look at Monte Carlo error per computation time standard multinomial resampling performs better. On the other hand, we see that classical systematic resampling remains to outperform other algorithms in terms of both speed and Monte Carlo error, while its butterfly counterpart shows a slightly worse performance. However, for very large number of particles, classical resampling algorithms become numerically unstable in single precision arithmetic and in this case butterfly resampling can become our only viable option. In Section 4.2, we compared resampling algorithms in the context of particle filters. In these experiments we used a simple model in order to facilitate closed form computations of estimated quantities. Our results suggest that butterfly multinomial resampling becomes faster especially in the context of adaptive resampling particle filters and becomes advantageous also in terms of MSE per computation time.

5.2. Future Work

Although butterfly resampling algorithm can be adjusted to fit better into communication structure imposed by CUDA architecture, its implementation is not straightforward and needs substantial effort compared to standard multinomial resampling. We have implemented the butterfly resampling scheme on GPU architecture but we posit that structure of butterfly graphs can be mapped exactly to hypercube or generalized hypercube architectures on which its performance gains would be more apparent. Butterfly resampling scheme is just a small fraction of endless possibilities offered by augmented resampling framework. For example, in case of grid architectures, banded matrices which impose interaction on neighbouring processors can be considered. A fruitful approach might be to consider the factorizations of $\mathbf{1}_N$ matrix into matrices that are adjacency matrices of a certain parameterized family of parallel architectures.

Apart from the communication of processors, another consideration in parallel implementation of particle filters is the movement of resampled particles in the memory. While it is possible to develop efficient algorithms that solves the problem of optimally moving particles in the memory, another approach would be to develop resampling algorithms that will produce ancestor vector that requires smaller cost of memory movements. Authors of [42] consider this problem and suggest a parallel algorithm to permute the ancestor vector so that the particles can be propagated in place, i.e. without needing a double buffer strategy.

APPENDIX A: IMPLEMENTATION DETAILS

In this chapter we present the implementation details of the various parts of resampling algorithms. We explain the design of our code and the motivations behind our design decisions. We present only the performance critical parts of the CUDA code since the whole code takes too much space and most of it consists of trivial details. In Section A.1, we describe the CUDA implementation of prefix sum computation and give code pieces. For numerical stability, instead of weights themselves we keep the logarithm of weights, so for addition of two weights we use the `log_sum_exp` function which takes two numbers a, b and returns $\log(\exp(a) + \exp(b))$. Since CUDA implementations of classical resampling algorithms are straightforward, we do not present them here. In Section A.2 we present implementation details for performance critical parts of butterfly resampling algorithm and in Section A.3 we present implementation details regarding the propagation of particles.

A.1. Implementation Details of Prefix Sum

Since prefix sum computation constitutes an essential component of the resampling algorithms here we give its detailed description with codes. In our implementations we used *work efficient scan* algorithm as given in [34, Chapter 9]. This implementation is called work efficient since it performs a total $\mathcal{O}(N)$ addition operations as opposed to $\mathcal{O}(N \log N)$ addition operations of a more straightforward approach while retainin $\mathcal{O}(\log N)$ time cost.

Work efficient scan algorithm consists of two main parts : a reduction phase in which an intermediate form of cumulative sums if computed and a distribution phase in partial sums are added to required positions. Both phases are implemented within a **for** loop. At each iteration of the **for** loop of the reduction phase, the number of active threads are halved and active threads operate on array elements seperated by a gap that increase at every iteration. We present CUDA implementation of reduction phase of work efficient scan algorithm in Table A.1. Here the variable `idx` is the index

of thread within the block, i.e. the built in variable `threadIdx.x`. Unusual way of indexing in this code decreases the thread divergence to a minimal level. At the end of reduction phase cumulative sums at the position 2^i for $i = 1, \ldots, \log_2 N$ are computed. We present CUDA implementation of distribution phase in Table A.2., again to avoid thread divergence we map the threads to array elements in an unusual way.

Table A.1. Reduction phase of work efficient scan algorithm.

```
// reduction phase
for (int stride = 1; stride <= blockDim.x; stride *= 2)
{
  int index = (idx + 1) * stride * 2 - 1;
  if (index < 2 * blockDim.x)
    cache[index] = log_sum_exp(cache[index], cache[index
      - stride]);
  __syncthreads();
}
```

In this implementation of scan algorithm a thread block operates on an array of double the size of thread block. Since we need to synchronize the threads in scan algorithm, we cannot perform scan on the array of the size greater than twice the maximum block size 1024. Let $\mathcal{I} = (I_1, \ldots, I_n)$ be a partition of the set $1, \ldots, N$ where each block I_i, $i = 1, \ldots, n$ are of size 2048 and $n = N/2048$. To perform prefix sum operations over all weights $w = (w^j)_{j=1}^N$, first we make a kernel call to perform prefix sum operation within each block $w^{I_i} = w^{I_i^{1:2048}}$ to get partial cumulative sums $W_i = (W_i^j = \sum_{k=1}^j w^{I_i^k})_{j=1}^{2048}$. To simplify the implementation, in a second kernel call, we use just one thread block. In this kernel, we compute cumulative sum of the array formed by the last elements of W_is, $v = (v^i = W_i^{2048})_{i=1}^n$. If $n \leq 2048$, we can perform this operation within one block, but if $n > 2048$ we cannot synchronize threads. So, instead of making another kernel call, we serialize the computation of multiple blocks of prefix sum operation on v, in the first iteration we compute cumulative sum of first

Table A.2. Distribution phase of work efficient scan algorithm.

```
// post reduction - distribute intermediate elements
for (int stride = blockDim.x / 2; stride > 0; stride /=
   2)
{
  int index = (idx + 1) * stride * 2 - 1;
  if (index + stride < 2 * blockDim.x)
    cache[index + stride] = log_sum_exp(cache[index +
       stride], cache[index]);
  __syncthreads();
}
```

2048 elements of v, in the second iteration we compute cumulative sum of next 2048 elements and add the last element of previous block to each element in the current block to get the cumulative sum of first 4096 elements and so on. Since we do not work on arrays of size greater than 2048^2 frequently, only one iteration suffices in most cases and even though we need to perform more iterations the overhead of kernel calls and control structures would be more compared to this partial serialization of the computation. After this step, we obtain a partial cumulative sum as in the reduction phase of work efficient scan algorithm and in a last kernel call, we add these partial sums to correct positions. We note that, this approach is needed only for the classical resampling methods, in which we need to compute the prefix sum over all weights array. In butterfly resampling scheme, at each stage of the resampling, only one kernel call to scan algorithm is enough since we use radix sequences having maximum radix less than 2048.

A.2. Implementation Details of Butterfly Resampling Methods

Efficient implementation of butterfly resampling algorithms on CUDA platform is not straightforward. Although block structure of butterfly resampling algorithm can

be mapped to hierarchical architecture of CUDA with ease, performance of resampling algorithm is largely affected by the strided memory accesses and extra kernel call overhead. However, these costs can be compensated by exploiting certain properties of butterfly resampling and a popular strategy for coalescing the global memory accesses known as tiling.

The first trick we utilize to speed up the butterfly resampling algorithm exploits the property that after a resampling stage, weights within a block become uniform, so instead of keeping all weights, we discard the multiple copies of the uniform weights and at each stage we perform cumulative sum operation on a smaller array. For a butterfly resampling algorithm parameterized with radix sequence $r = (r_1, \ldots, r_m)$ where $\prod_{k=1}^{m} r_k = N$, at the first stage, we need to compute cumulative sum of N weights. After resampling with first radix r_1, we obtain a weight vector of N/r_1 distinct weights. So at the next stage we need to compute cumulative sum of N/r_1 weights and carefully map the resulting cumulative sums to correct particles. Continuing in this way, at the kth stage we compute a cumulative sum over a weights array of size $N/(r_1 \ldots r_{k-1})$. This strategy substantially reduces the computation cost associated to prefix sum operation. CUDA implementation of this idea is presented in

Another performance critical part of the butterfly resampling algorithm is the mapping of the particles to threads. The straightforward approach is to use the modular congruence relations as defined in equations 3.6 and 3.7. For butterfly resampling algorithm parameterized with radix sequence $r = (r_1, \ldots, r_m)$, using Equation 3.7, we map the ith particle at the kth stage to the thread block indexed by $\mathcal{I}_{r,k,i}$. Threads in this block, access global memory locations that are $r_1 \ldots r_{k-1}$ apart from each other. So, using Equation 3.6, the ith particle will be mapped to the thread having index $(i - i_{r,k,\mathcal{I}_{r,k,i}})/(r_1 \ldots r_{k-1}) + 1$ within the block. Strided memory accesses of this approach can cause bad utilization of high memory bandwidth of the GPU. In order to avoid this, we need to coalesce the global memory accesses.

Suppose we use a uniform radix sequence where $r_k = R$ for all $k = 1, \ldots, m$. Examining the patterns of above mapping strategy by considering base R representa-

Table A.3. Shrinking weights array in butterfly resampling.

```
int w_size = n_particles;
for (int j = 0; j < n_steps; j++)
{
  // compute cumulative sums of strided blocks
  scan_cum_sum_eff<<<w_size/block_sizes[j], block_sizes[j
     ]/2, block_sizes[j]*sizeof(float)>>>(weights,
     next_weights);

  // resample
  bfmnr_kernel<<<n_particles/block_sizes[j], block_sizes[
     j], block_sizes[j]*sizeof(float)>>>(weights,
     prng_states, ancestors, bw_lengths[j], positions[j])
     ;

  swapf(&weights, &next_weights);
  w_size /= block_sizes[j];
}
```

tion of particle indices, we see a pattern. Let $i = d_m d_{m-1} \dots d_1$ in m-digits base R representation where $d_k \in \{0, \dots, R-1\}$ for all $k = 1, \dots, m$. Then at the kth stage of butterfly resampling, the ith particle is mapped to the d_kth thread of the thread block with index $I = d_m \dots d_{k-1} d_{k+1} \dots d_1$ in base R representation. In a scenario where threads are indexed linearly, this suggests that particle with index $d_m d_{m-1} \dots d_1$ in base R is mapped to the thread with index $d_m \dots d_{k+1} d_{k-1} \dots d_1 d_k$ in base R. Based on this observation, we carry out the mapping of particles to threads by using simple bit manipulations. CUDA implementation of butterfly resampling procedure using these bit manipulations is given in Table A.4. Even more, this observations enables us to coalesce the memory accesses by permuting the particles in memory at every stage

of butterfly resampling algorithm.

Table A.4. Butterfly resampling kernel with strided accesses.

```
local_cum_sum[t_idx] = weights[global_idx >> bfindexpos];
    // cumulative weights are already computed
curandState local_state = prng_states[idx]; // get PRNG
   state

__syncthreads();

float u =  local_cum_sum[blockDim.x - 1] + logf(
   curand_uniform(&local_state)); // generate uniform
   random number in (0, local_W_end]
prng_states[idx] = local_state; // update PRNG state

// locate position of u in local_cum_weights and compute
   the global index of the ancestor
int local_ancestor = ancestors[dev_block_bitperm(b_idx,
   radixlog2 , bfindexpos) | (lower_bound(u,
   local_cum_sum, blockDim.x) << bfindexpos)];

__syncthreads();

ancestors[global_idx] = local_ancestor; // strided write
```

Above observation shows that the partitioning structure of the butterfly algorithm can be thought as a permutation of the digits of the particle indices at base R. We obtained those permutations by taking each digit to the rightmost position, however different permutation strategies are possible and can lead to utilization of coalesced memory accesses. To understand how we can achieve coalescence of memory accesses,

suppose we populated the particles into a matrix of size $N/R \times R$. Considering linear indices in row major order and assuming base R representations of indices, the particle with index $d_m d_{m-1} \dots d_1$ resides at the position $(d_m \dots d_2, d_1)$. Transposing the matrix and linearizing we see that the particle with index $d_m d_{m-1} \dots d_1$ comes to the position $d_1 d_m \dots d_2$. Mapping these particles linearly to the threads, we map particle $d_1 d_m \dots d_2$ to thread $d_m d_{m-1} \dots d_1$ which correspond to a different blocking structure than the previous ordering of particles.

Table A.5. Tiled matrix transposition.

```
extern __shared__ REAL tile[];
int tx = threadIdx.x, ty = threadIdx.y; // thread x,y
   indices
int gix = tx + blockIdx.x * blockDim.x, giy = ty +
   blockIdx.y * blockDim.y; // global input matrix x,y
   indices
int gox = tx + blockIdx.y * blockDim.y, goy = ty +
   blockIdx.x * blockDim.y; // global output matrix x,y
   indices

// coalesced read from global array and strided write to
   shared memory
tile[ty + tx * blockDim.y] = input_matrix[gix + giy *
   n_columns]; // linearized row major indices
__syncthreads();

// coalesced write to output array
output_matrix[gox + goy * n_rows] = tile[tx + ty *
   blockDim.x];
```

Applying this tranposition procedure repeatedly, we apply a cyclic permutation

on the digits of base R representations of particle indices and mapping particles to threads linearly, we get a different blocking structure at each stage. Upside of this strategy is that matrix transposition operation can be performed with coalesced global memory accesses on GPUs via a technique called *tiling* [34]. CUDA implementation of tile matrix transposition is presented in Table A.5.

We facilitate this strategy in butterfly resampling as follows: after the resampling operation at the kth stage, we form N/r_k matrix from the particles, transpose this matrix and linearize it. Assuming matrices are stored in row major order, formation and linearization of matrices is not performed actually, it is carried out implicitly by just changing the indexing. Although this strategy can provide speedup by coalescing memory accesses, it imposes an extra kernel call and memory space overhead since tiling can not be performed by inplace computations and complicates the implementation procedure with additional consideration on the indexing of weights. We present CUDA implementation of butterfly resampling with coalesced memory accesses in Table A.6.

A.3. Propagation of Particles in Particle Filter Implementations

Considering the low latency of global memory accesses of GPU architecture, propagation of the particles is a principal part of the GPU implementations of particle filters. As we have pointed out earlier in Section 2.1, one way to ensure in-place propagation of particles is to make sure that ancestors $a = (a^1, \ldots, a^N)$ satisfy the condition that for all $i = 1, \ldots, N$ if $a^j = i$ for some $j = \{1, \ldots, N\}$ then $a^i = i$. Since random output of resampling algorithms does not satisfy this condition necessarily, authors of [42] propose a parallel algorithm that can permute the ancestor vector so that it satisfy this condition. However, we do not adopt this approach here. Since the full interaction structure of classical resampling algorithms cannot be mapped to constrained communication structure of GPU architecture, we have to use a double buffer strategy for the propagation of particles in a particle filter that employ classical resampling algorithms. In Table A.7., we present GPU implementation of multinomial resampling with double buffer strategy.

Table A.6. Butterfly resampling kernel with coalesced memory accesses.

```
local_cum_sum[t_idx] = weights[idx&(~(0xFFFFFFFF<<
    weightindexpos))];
curandState local_state = prng_states[idx]; // get PRNG
    state

float u =  local_cum_sum[blockDim.x - 1] + logf(
    curand_uniform(&local_state)); // generate uniform
    random number in (0, local_W_end]
prng_states[idx] = local_state; // update PRNG state

// locate position of u in local_cum_weights and compute
    the global index of the ancestor
int local_ancestor = ancestors[lower_bound(u,
    local_cum_sum, blockDim.x) + b_idx * blockDim.x];

__syncthreads();

ancestors[idx] = local_ancestor; // coalesced write
```

Local interaction structure of butterfly resampling scheme enables in-place propagation of particles. Since particle blocks do not intersect with each other and threads within a block can synchronize at a barrier, hazardous behaviour of the race conditions can be avoided. After resampling an ancestor within its local group, each thread reads its ancestor particle into a local buffer and all threads within the block synchronize. This synchronization step ensures that no write operation is performed before all the resampled ancestors are read. Finally, each thread writes the content of its local buffer to associated memory location. CUDA implementation of this approach is given in Table A.8. In-place propagation of particles requires the half of the memory space

Table A.7. Multinomial resampling kernel with double buffer implementation.

```
int idx = threadIdx.x + blockIdx.x * blockDim.x;

float cum_sum = cum_weights[n_sample - 1];
float u =  cum_sum + logf(curand_uniform(&prng_states[idx
   ])); // generate uniform random number in (0, W_N]

// locate position of u in cum_weights
int ancestor = lower_bound(u, cum_weights, n_sample);
new_particles[idx] = old_particles[ancestor];

weights[idx] = cum_sum - logf((float) n_sample); // set
   all weights to W_N/N
```

reqired by the double buffer approcah. However, in practice this approach should be considered carefully. In high dimensional particle regimes, moving particles at every stage can be prohibitively expensive.

Table A.8. Butterfly resampling kernel with in-place propagation.

```
local_cum_sum[t_idx] = cum_weights[global_idx >>
   bfindexpos]; // cumulative weights are already
   computed
curandState local_state = prng_states[idx]; // get PRNG
   state

__syncthreads();

float u =  local_cum_sum[blockDim.x - 1] + logf(
   curand_uniform(&local_state)); // generate uniform
   random number in (0, local_W_end]
prng_states[idx] = local_state; // update PRNG state

// locate position of u in local_cum_weights and compute
   the global index of the ancestor
float local_ancestor_particle = particles[
   dev_block_bitperm(b_idx, radixlog2 , bfindexpos) | (
   lower_bound(u, local_cum_sum, blockDim.x) <<
   bfindexpos)];

__syncthreads();

particles[global_idx] = local_ancestor_particle;
```

REFERENCES

1. Kalman, R. E., "A New Approach to Linear Filtering and Prediction Problems", *Journal of Fluids Engineering*, Vol. 82, No. 1, pp. 35–45, 1960.

2. Kalman, R. E. and R. S. Bucy, "New Results in Linear Filtering and Prediction Theory", *Journal of Fluids Engineering*, Vol. 83, No. 1, pp. 95–108, 1961.

3. Julier, S. J. and J. K. Uhlmann, "A New Extension of the Kalman Filter to Nonlinear Systems", *International Symposium on Aerospace/Defense Sensing, Simulation and Controls*, Vol. 3, pp. 182–193, 1997.

4. Julier, S. J. and J. K. Uhlmann, "Unscented Filtering and Nonlinear Estimation", *Proceedings of the IEEE*, Vol. 92, No. 3, pp. 401–422, 2004.

5. Doucet, A., N. De Freitas and N. Gordon, *Sequential Monte Carlo Methods in Practice*, Springer, 2001.

6. Arulampalam, M., S. Maskell, N. Gordon and T. Clapp, "A Tutorial on Particle Filters for Online Nonlinear/Non-Gaussian Bayesian Tracking", *Transactions on Signal Processing*, Vol. 50, No. 2, pp. 174–188, Feb. 2002.

7. Doucet, A. and A. M. Johansen, "A Tutorial on Particle Filtering and Smoothing: Fifteen Years Later", *Handbook of Nonlinear Filtering*, Vol. 12, pp. 656–704, 2009.

8. Cappé, O., S. J. Godsill and E. Moulines, "An Overview of Existing Methods and Recent Advances in Sequential Monte Carlo", *Proceedings of the IEEE*, Vol. 95, No. 5, pp. 899–924, 2007.

9. Kunsch, H. R., "Particle Filters", *Bernoulli*, Vol. 19, No. 4, pp. 1391–1403, 09 2013.

10. Gordon, N. J., D. J. Salmond and A. F. Smith, "Novel Approach to Nonlinear/Non-Gaussian Bayesian State Estimation", *IEE Proceedings F (Radar and Signal Processing)*, Vol. 140, pp. 107–113, IET, 1993.

11. Stewart, L. and P. McCarty Jr, "Use of Bayesian Belief Networks to Fuse Continuous and Discrete Information for Target Recognition, Tracking, and Situation Assessment", *Aerospace Sensing*, pp. 177–185, International Society for Optics and Photonics, 1992.

12. Liu, J. S. and R. Chen, "Blind Deconvolution via Sequential Imputations", *Journal of the American Statistical Association*, Vol. 90, No. 430, pp. 567–576, 1995.

13. Pitt, M. K. and N. Shephard, "Filtering via Simulation: Auxiliary Particle Filters", *Journal of the American Statistical Association*, Vol. 94, No. 446, pp. 590–599, 1999.

14. Carpenter, J., P. Clifford and P. Fearnhead, "Improved Particle Filter for Nonlinear Problems", *IEE Proceedings-Radar, Sonar and Navigation*, Vol. 146, No. 1, pp. 2–7, 1999.

15. Gilks, W. R. and C. Berzuini, "Following a Moving Target—Monte Carlo Inference for Dynamic Bayesian Models", *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, Vol. 63, No. 1, pp. 127–146, 2001.

16. Doucet, A., N. De Freitas, K. Murphy and S. Russell, "Rao-Blackwellised Particle Filtering for Dynamic Bayesian Networks", *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence*, pp. 176–183, Morgan Kaufmann Publishers Inc., 2000.

17. Musso, C., N. Oudjane and F. Le Gland, "Improving Regularised Particle Filters", *Sequential Monte Carlo Methods in Practice*, pp. 247–271, Springer, 2001.

18. Crisan, D. and A. Doucet, "Convergence of Sequential Monte Carlo Methods",

Signal Processing Group, Department of Engineering, University of Cambridge, Technical Report, Vol. 1, 2000.

19. Chopin, N., "Central Limit Theorem for Sequential Monte Carlo Methods and Its Application to Bayesian Inference", *Annals of Statistics*, pp. 2385–2411, 2004.

20. Douc, R. and E. Moulines, "Limit Theorems for Weighted Samples with Applications to Sequential Monte Carlo Methods", *ESAIM: Proceedings*, Vol. 19, pp. 101–107, EDP Sciences, 2007.

21. Bickel, P., B. Li, T. Bengtsson *et al.*, "Sharp Failure Rates for the Bootstrap Particle Filter in High Dimensions", *Pushing the Limits of Contemporary Statistics: Contributions in Honor of Jayanta K. Ghosh*, pp. 318–329, Institute of Mathematical Statistics, 2008.

22. Douc, R., E. Moulines and J. Olsson, "Long-term Stability of Sequential Monte Carlo Methods Under Verifiable Conditions", *arXiv preprint*, 2012, `http://arxiv.org/abs/1203.6898`, [Accessed January 2015].

23. Whiteley, N., "Stability Properties of Some Particle Filters", *The Annals of Applied Probability*, Vol. 23, No. 6, pp. 2500–2537, 2013.

24. Rebeschini, P. and R. Van Handel, "Can Local Particle Filters Beat the Curse of Dimensionality?", *arXiv preprint*, 2013, `http://arxiv.org/abs/1301.6585`, [Accessed January 2015].

25. Beskos, A., D. Crisan and A. Jasra, "On the Stability of Sequential Monte Carlo Methods in High Dimensions", *The Annals of Applied Probability*, Vol. 24, No. 4, pp. 1396–1445, 2014.

26. Savage, J. E., *Models of Computation: Exploring the Power of Computing*, Addison-Wesley Longman Publishing Co., Inc., 1st edn., 1997.

27. Forum, M. P., *MPI: A Message-Passing Interface Standard*, Tech. rep., 1994.

28. Asanovic, K., R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel and K. Yelick, "A View of the Parallel Computing Landscape", *Communications of the ACM*, Vol. 52, No. 10, pp. 56–67, Oct. 2009.

29. Hwu, W.-m. W., *GPU Computing Gems Jade Edition*, Morgan Kaufmann Publishers Inc., 1st edn., 2011.

30. Buck, I., T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston and P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware", *ACM Transactions on Graphics*, Vol. 23, No. 3, pp. 777–786, Aug. 2004.

31. Nickolls, J., I. Buck, M. Garland and K. Skadron, "Scalable Parallel Programming with CUDA", *Queue*, Vol. 6, No. 2, pp. 40–53, Mar. 2008.

32. Stone, J. E., D. Gohara and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems", *Computing in Science and Engineering*, Vol. 12, No. 3, pp. 66–73, May 2010.

33. Sanders, J. and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Addison-Wesley Professional, 1st edn., 2010.

34. Kirk, D. B. and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann Publishers Inc., 1st edn., 2010.

35. Lee, A., C. Yau, M. B. Giles, A. Doucet and C. C. Holmes, "On the Utility of Graphics Cards to Perform Massively Parallel Simulation of Advanced Monte Carlo Methods", *Journal of Computational and Graphical Statistics*, pp. 769–789, 2010.

36. Suchard, M. A., Q. Wang, C. Chan, J. Frelinger, A. Cron and M. West, "Understanding GPU Programming for Statistical Computation: Studies in Massively

Parallel Massive Mixtures", *Journal of Computational and Graphical Statistics*, Vol. 19, No. 2, pp. 419–438, 2010.

37. Suchard, M. A., S. E. Simpson, I. Zorych, P. Ryan and D. Madigan, "Massive Parallelization of Serial Inference Algorithms for a Complex Generalized Linear Model", *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, Vol. 23, No. 1, p. 10, 2013.

38. Murray, L. M., "Bayesian State-space Modelling on High-performance Hardware Using LibBi", *arXiv preprint*, 2013, `http://arxiv.org/abs/1306.3277`, [Accessed January 2015].

39. Guldas, H. and A. T. Cemgil, "Rasgele Örnekleme ile Düşük Rank Matris Tamamlama (Low Rank Matrix Completion via Random Sampling)", *IEEE Signal Processing and Communications Applications Conference*, 2013.

40. Doucet, A., N. De Freitas and N. Gordon, "An Introduction to Sequential Monte Carlo Methods", *Sequential Monte Carlo Methods in Practice*, pp. 3–14, Springer, 2001.

41. Doucet, A., S. Godsill and C. Andrieu, "On Sequential Monte Carlo Sampling Methods for Bayesian Filtering", *Statistics and Computing*, Vol. 10, No. 3, pp. 197–208, 2000.

42. Murray, L. M., A. Lee and P. E. Jacob, "Parallel Resampling in the Particle Filter", *arXiv preprint*, 2014, `http://arxiv.org/abs/1301.4019`, [Accessed January 2015].

43. Whiteley, N., A. Lee and K. Heine, "On the Role of Interaction in Sequential Monte Carlo Algorithms", *arXiv preprint*, 2013, `http://arxiv.org/abs/1309.2918`, [Accessed January 2015].

44. Bolic, M., P. M. Djuric and S. Hong, "Resampling Algorithms and Architectures

for Distributed Particle Filters", *IEEE Transactions on Signal Processing*, Vol. 53, pp. 2442–2450, 2004.

45. Lee, A. and N. Whiteley, "Forest Resampling for Distributed Sequential Monte Carlo", *arXiv preprint*, 2014, `http://arxiv.org/abs/1406.6010`, [Accessed January 2015].

46. Brun, O., V. Teuliere and J.-M. Garcia, "Parallel Particle Filtering", *Journal of Parallel Distributed Computing*, Vol. 62, No. 7, pp. 1186–1202, Jul. 2002.

47. Míguez, J., M. F. Bugallo and P. M. Djurić, "A New Class of Particle Filters for Random Dynamic Systems with Unknown Statistics", *EURASIP Journal on Applied Signal Processing*, Vol. 2004, pp. 2278–2294, 2004.

48. Li, C., Q. Honglei and X. Juhong, "Distributed Genetic Resampling Particle Filter", *Advanced Computer Theory and Engineering (ICACTE), 2010 3rd International Conference on*, Vol. 2, pp. V2–32, IEEE, 2010.

49. Paige, B., F. Wood, A. Doucet and Y. W. Teh, "Asynchronous Anytime Sequential Monte Carlo", *arXiv preprint*, 2014, `http://arxiv.org/abs/1407.2864`, [Accessed January 2015].

50. Vergé, C., C. Dubarry, P. Del Moral and E. Moulines, "On Parallel Implementation of Sequential Monte Carlo Methods: the Island Particle Model", *Statistics and Computing*, pp. 1–18, 2013.

51. Heine, K., N. Whiteley, A. T. Cemgil and H. Guldas, "Butterfly Resampling: Asymptotics for Particle Filters with Constrained Interactions", *arXiv preprint*, 2014, `http://arxiv.org/abs/1411.5876`, [Accessed January 2015].

52. Guldas, H., A. T. Cemgil, N. Whiteley and K. Heine, "A Practical Introduction to Butterfly and Adaptive Resampling in Sequential Monte Carlo", *The 17th IFAC Symposium on System Identification, SYSID 2015*, 2015, manuscript submitted

for publication.

53. Koller, D. and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*, The MIT Press, 2009.

54. NVIDIA Corporation, *NVIDIA CUDA C Programming Guide v6.5*, August 2014.

55. Sengupta, S., M. Harris, Y. Zhang and J. D. Owens, "Scan Primitives for GPU Computing", *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '07, pp. 97–106, Eurographics Association, 2007.

56. Olsson, J., O. Cappé, R. Douc and r. Moulines, "Sequential Monte Carlo Smoothing with Application to Parameter Estimation in Nonlinear State Space Models", *Bernoulli*, Vol. 14, No. 1, pp. 155–179, 02 2008.

57. Douc, R. and O. Cappé, "Comparison of Resampling Schemes for Particle Filtering", *Proceedings of the 4th International Symposium on Image and Signal Processing and Analysis, 2005. ISPA 2005.*, pp. 64–69, IEEE, 2005.

58. Hol, J. D., T. B. Schon and F. Gustafsson, "On Resampling Algorithms for Particle Filters", *Nonlinear Statistical Signal Processing Workshop, 2006 IEEE*, pp. 79–82, IEEE, 2006.

59. Devroye, L., *Non-Uniform Random Variate Generation*, Springer-Verlag, 1986.

60. Kitagawa, G., "Monte Carlo Filter and Smoother for Non-gaussian Nonlinear State Space Models", *Journal of Computational and Graphical Statistics*, Vol. 5, No. 1, pp. 1–25, 1996.

61. Oppenheim, A. V., R. W. Schafer and J. R. Buck, *Discrete-time Signal Processing (2nd Ed.)*, Prentice-Hall, Inc., 1999.

62. Dempster, A. P., N. M. Laird and D. B. Rubin, "Maximum Likelihood from In-

complete Data via the EM Algorithm", *Journal of the Royal Statistical Society, Series B*, Vol. 39, No. 1, pp. 1–38, 1977.

63. Fort, G. and E. Moulines, "Convergence of the Monte Carlo Expectation Maximization for Curved Exponential Families", *Annals of Statistics*, Vol. 31, No. 4, pp. 1220–1259, 08 2003.

64. Hull, J. and A. White, "The Pricing of Options on Assets with Stochastic Volatilities", *The Journal of Finance*, Vol. 42, No. 2, pp. 281–300, 1987.