PARALLEL RESAMPLING FOR REAL-TIME SEQUENTIAL MONTE CARLO

by

Alper Kamil Bozkurt

B.S., Computer Engineering, Boğaziçi University, 2015

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

2018

ACKNOWLEDGEMENTS

With my deepest gratitude, I would first like to thank my thesis supervisor Prof. Ali Taylan Cemgil for all his support and guidance throughout the course of this research. I would also like to thank Prof. Can Özturan and Assist. Prof. Sinan Yıldırım for participating in my thesis jury.

I would especially like to thank Fahri Serhan Daniş for his help during our collaboration. Additionally, I would like to thank all my friends and colleagues in Perceptual Intelligence Laboratory for their continuous assistance during this research, namely, Necati Cihan Camgöz, Taha Yusuf Ceritli, Barış Evrim Demiröz, Ahmet Alp Kındıroğlu, Hazal Koptagel, Mehmet Burak Kurutmaz, Oğulcan Özdemir, Recep Doğa Siyli, and Umut Şimşekli.

I would like to thank my dear friends in the computer engineering department for their support and valuable friendship, namely, Helin Ece Akgül, Gökcan Çantalı, Metehan Doyran, Hasan Ferit Enişer, Orhan Ermiş, Gürkan Gür, Mehmet Can Güven, Hakan Selvi, Nefise Gizem Yağlıkçı, and Yiğit Yıldırım.

Finally, I would like to thank my parents and my sisters for their endless love, support and encouragement throughout my life. I could not have done this without them.

ABSTRACT

PARALLEL RESAMPLING FOR REAL-TIME SEQUENTIAL MONTE CARLO

Sequential Monte Carlo (SMC) methods are well known and widely used for state estimation in nonlinear/non-Gaussian dynamical systems. However due to the heavy computational requirements, they may not satisfy the real-time constraints in many applications requiring a large number of samples. By means of parallel implementation, real-time tasks such as online filtering can be achieved. However, the resampling stage in SMC methods requires sample interaction, hence it is not trivial to parallelize. In this thesis, we first provide a standard way to parallelize resampling algorithms, and discuss the issues arising from its implementation. We then propose a parallel resampling algorithm, resampling with butterfly communications (RBC), inspired by butterfly resampling previously described in the literature. Our aim is to eliminate the important bottlenecks of the standard approach such as communication overhead and load imbalance by imposing constraints on the communication pattern. We conducted experiments on different parallel computing architectures including computer clusters, and GPUs. We compared the RBC algorithm with the standard approach in terms of execution time and accuracy. We found that the RBC algorithm outperforms the standard approach on computer clusters and GPUs, and successfully achieves high speed and accuracy in exchange for negligible loss of effective sample size.

ÖZET

GERÇEK ZAMANLI ARDIŞIK MONTE CARLO ÖRNEKLEYİCİLERİ İÇİN PARALEL YENİDEN ÖRNEKLEME

Ardışık Monte Carlo (SMC) yöntemleri, doğrusal ve Gaussian olmayan dinamik sistemlerde iyi bilinen ve çokça kullanılan yöntemlerdir. Ancak, ağır hesaplama gereklilikleri nedeniyle, çok sayıda örnek gerektiren birçok uygulamada gerçek zamanlı kısıtlara uymayabilirler. Paralelleştirme sayesinde, çevrimiçi süzgeçleme gibi gerçek zamanlı görevler gerçekleştirilebilir. Fakat, SMC yöntemlerinde yeniden örnekleme aşaması örnek etkileşimini gerektirir ve dolayısıyla paralelleştirilmesi kolay değildir. Bu tezde, ilk olarak, yeniden örnekleme algoritmalarını paralel hale getirmenin standart yöntemini sunuyoruz ve bu yöntemin gerçekleşiminde açığa çıkan sorunları tartışıyoruz. Daha sonra, literatürde daha önce tarif edilen kelebek yeniden örneklendirmesinden esinlenerek, kelebek iletişimi (RBC) ile yeniden örnekleme yapan paralel bir yeniden örnekleme algoritması öneriyoruz. Amacımız, iletişim örüntüsüne kısıtlar koyarak, iletişim yükü ve yük dengesizliği gibi standart yaklaşımın önemli darboğazlarını ortadan kaldırmaktır. Deneylerimizi bilgisayar grupları ve GPU'lar dahil olmak üzere farklı paralel hesaplama mimarileri üzerinde yaptık. RBC algoritmasını standart yaklaşımla, yürütüm süresi ve doğruluk açısından karşılaştırdık. RBC algoritmasının, bilgisayar kümeleri ile GPU'lar üzerinde standart yaklaşımı geride bıraktığını ve etkin örneklem büyüklüğünün ihmal edilebilir kaybına karşılık, yüksek hızı ve doğruluğu başarılı bir şekilde elde ettiğini gördük.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS

A_t^i	Ancestor index of the particle with index i at time step $t+1$
$A_{i:t}^i$	Ancestor lineage of the particle with index i from time step 1 to t
$\mathbb{A}$	Set of Markov transition matrices
$\mathrm{Cat}(\cdot)$	Categorical Distribution
$\mathbb{E}[\cdot]$	Expectation
$f(\cdot)$	Transition density
$\mathcal{F}(\cdot)$	Discrete probability distribution used in resampling
g	Gap between consecutive transmission or reception operations
$g(\cdot)$	Observation density
I_d	$d \times d$ identity matrix
$I_{\mathrm{SIS}}(\varphi(x_{1:t}))$	Sequential importance sampling approximation of the expectation of a test function $\varphi(x_{1:t})$
L	Latency of a network
L_t^k	Number of shortage or surplus particles of the processing element with index k at time step t
M	Number of particles assigned to a single processing element
M_t^k	Offspring number of the processing element with index k at time step t
$\mathrm{Mult}(\cdot,\cdot)$	Multinomial Distribution
N	Number of particles
N_t^{eff}	Effective Sample size at time step t
N_t^i	Offspring number of the particle with index i at time step t
o	Overhead of transmitting and receiving a message
P	Number of processing elements
$q(x_{1:t})$	Proposal distribution
T	Resampling time
U_{r_k}	$r_k \times r_k$ matrix having $1/r_k$ as every entry
W_t^i	Weight of the particle with index i at time step t

w_t^i	Normalized weight of the particle with index i at time step t
X_t^i	Particle with index i at time step t
$X_{1:t}^i$	Particle trajectory with index i from time step 1 to t
$\mathcal{X}^n$	Product Space
Z_t	Normalizing constant at time step t
$\hat{Z}_t$	Importance sampling approximation of the normalizing constant at time step t
α	Markov transition matrix
$\pi(x_{1:t})$	Target distribution
$\hat{\pi}(x_{1:t})$	Importance sampling approximation of the target distribution
$\phi(x_{1:t})$	Probability measure on $\mathcal{X}^n$

LIST OF ACRONYMS/ABBREVIATIONS

AR	Augmented Resampling
BR	Butterfly Resampling
BLE	Bluetooth Low Energy
BSP	Bulk Synchronous Parallel
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DRAM	Dynamic Random-Access Memory
ESS	Effective Sample Size
GPGPU	General Purpose Computing on Graphics Processing Units
GPU	Graphics Processing Unit
HMM	Hidden Markov Model
HPC	High Performance Computing
IS	Importance Sampling
LGSSM	Linear Gaussian State Space Model
LM	Local Memory
MCMC	Markov Chain Monte Carlo
MIMD	Multiple Instruction, Multiple Data
MSE	Mean squared errors
PE	Processing Element
PRAM	Parallel Random-Access Machine
RBC	Resampling with Butterfly Communications
RNA	Resampling Algorithms with Non-Proportional Allocation
RPA	Resampling Algorithms with Proportional Allocation
RSSI	Received Signal Strength Indicator
SIS	Sequential Importance Sampling
SM	Streaming Multiprocessor
SMC	Sequential Monte Carlo
SIMD	Single Instruction, Multiple Data

SPMD	Single Program, Multiple Data
SSM	State Space Model

1. INTRODUCTION

Real-time inference from sequentially arriving data is required by many problems faced in domains as diverse as control systems, computer vision and finance. The general state space models (SSM) allow us to model the streaming data and perform inference. Inference in SSMs is basically the construction of a posterior distribution using all the available data. In linear Gaussian SSMs (LGSSM), the posterior distribution can be computed using Kalman techniques [1], and if the model has a finite state space, hidden Markov Model (HMM) methods [2] (forward and backward methods) can provide the required posterior distributions. Unfortunately, most of the real-world data are complex, highly nonlinear, and they have non-Gaussian characteristics. The assumptions of Kalman and HMM methods does not fit with the real-world data, and therefore they perform very poorly. In fact, except in a few simple cases, including LGSSMs and finite state HMMs, it is not possible to obtain closed-form expression for the posterior distribution [3]. Thus it is necessary to employ numerical methods to approximate the required distributions.

Sequential Monte Carlo (SMC) methods encompass a large set of techniques that approximate a sequence of posterior distributions. Unlike the previously proposed approximation methods such as the extended Kalman filter [4], SMC methods do not rely on any linearization technique or other assumptions to make sure the tractability [5]. SMC methods sequentially construct an approximation to the posterior distribution of the latent variables conditioned on all the available data, via weighted samples randomly. Initially SMC methods were used to perform filtering tasks in time-varying systems such as object tracking [3], then they were generalized and studied theoretically [6].

With the flexibility combined with the increasing computation power, SMC methods have been widely used in diverse areas of science including engineering, economics and biology [3]. Many SMC instances have been introduced for fighting with the degeneracy [7], for reducing the variance of estimates [8], and for smoothing tasks [9].

Moreover a new Markov chain Monte Carlo (MCMC) approach employs SMC to build proposal distributions [10]. More recently a novel variational approximating family combining variation inference and SMC methods has been introduced [11].

1.1. Motivation

Despite their flexibility, SMC methods are computationally expensive. The time complexity of SMC methods grows linearly with the number of samples to be used. In addition, the required number of samples increases with the dimensionality of the target distribution and the discrepancy between the target and the proposal distributions. Therefore, many real-world SMC applications need a large number of samples to provide good approximations for the target distribution.

Moreover, SMC methods are widely used in many computationally demanding algorithms. For example, the SMC-based implementations of the probability hypothesis density (PHD) filter [12, 13], use particles to approximate the multi-target filtering density. SMC-PHD filters can be very time consuming, and thus impractical, especially when the number of targets to be tracked is large. Another example is particle MCMC [10], which makes Bayesian inference possible for numerous statistical models. Particle MCMC methods run an SMC algorithm in each step of an MCMC algorithm, thus the computational cost is considerably high.

Fortunately, SMC methods are convenient for parallel implementation. The samples can be easily assigned to different processing elements (PEs), and each sample can be propagated and weighted independently without requiring any communication between PEs. The only stage in SMC methods requiring sample interaction is resampling, which is crucial for the stability. Resampling redistributes the samples in such a way that the samples with large weights are replicated and the samples with small weights are eliminated. Resampling is evidently the bottleneck in parallel implementation of SMC methods since all the samples must be combined in this stage.

There are several aspects that must be taken into account when designing a parallel resampling algorithm. First of all, the characteristics of parallel computing architectures such as interconnection networks and memory hierarchy could affect the execution time significantly, therefore they must be carefully considered. Second, the communication overhead and the load imbalance among PEs could easily degrade the performance. Therefore, a convenient parallel resampling algorithm should minimize the communication time and achieve a good load balance, at the same time, it should keep the effective sample size (ESS) reasonably high to ensure stability and high accuracy.

1.2. Approach and Contributions

In this thesis, we first provide a brief overview of the aspects of parallel computing architectures that play a critical role in designing parallel programs. We then describe a standard approach of parallelization of resampling algorithms, and we provide mathematical expressions for the execution time on different computing environments. After that, we propose a novel parallel resampling algorithm, resampling with butterfly communications (RBC), inspired by butterfly resampling described in [14, 15].

Our main contribution in this thesis is the proposed RBC algorithm. The RBC algorithm imposes constraints on the communication patterns of the PEs such that the communication overhead caused by a large number of messages or high latency memory accesses is prevented. In RBC, PEs are grouped in pairs according to the butterfly structure [14], and each PE can communicate with only the other PE in the same group. We show that the RBC algorithm is scalable, and it keeps the ESS above a reasonable threshold thereby guaranteeing stability. Furthermore, the RBC algorithm is parameterized by the number of communicating pairs, thus the tradeoff between performance and stability can be adaptively balanced.

We conducted experiments on different parallel computing architectures including computer clusters and GPUs. We compared the RBC algorithm with the standard approach in terms of execution time, resampling time, effective sample size and ac-

curacy. We found that the RBC algorithm is superior to the standard approach on clusters and GPUs, and successfully achieves high speed and accuracy in exchange for negligible loss of effective sample size.

The rest of the thesis is organized as follows. Chapter 2 gives an overview of existing parallel resampling techniques. In Chapter 3, we discuss the characteristics of widely used parallel computing architectures, and in Chapter 4 we outline a generic SMC method. A standard parallel resampling algorithm is described in Chapter 5 while the proposed RBC algorithm is explained in Chapter 6. Our experiment design and results are provided in Chapter 7. Finally, Chapter 8 draws conclusions.

2. RELATED WORK

SMC methods approximate the posterior distributions using weighted point mass samples called particles. Particles are propagated according to a state transition model, and a weight is assigned to each particle and updated with the observations. The common approach in parallelization of resampling in the literature is to divide the particles into distinct subsets, and to assign each subset to a PE. In this way, PEs are able to independently update and propagate the particles in their local subsets. However, resampling requires the weight information of all the particles, and may result in a highly unbalanced subsets, that is, some subsets could have a large number of particles while others only have few particles. Furthermore, both gathering the weight information and the particle transmission require communication. Therefore, parallelization of the resampling stage constitutes the main objective in designing parallel SMC methods.

2.1. Distributed Implementation of Resampling

In early efforts such as [16], the resampling stage is performed locally, that is to say, each PE resamples only the particles in its subset. This approach does not require any communication between PEs, however it has a significant drawback. The weights of the particles in all the subsets but one converges to zero very fast, and the subsets with zero weighted particles do not have any effect on the result. In other words, the parallelization does not provide any improvement in this scheme. In [17], two additional resampling schemes are proposed to solve this stability problem. In the first scheme, each PE sends their local particles to a master PE, then the master PE performs resampling and redistributes the particles to all PEs. The major disadvantage of this approach is that the transmission of the particles introduces a communication overhead, and overloads the master PE. The second scheme alleviates the overhead by compressing the replicated particles, but the compression ratio may not be good enough for many applications.

Another work [18] introduces two categories of parallel resampling algorithms: resampling algorithms with proportional allocation (RPA) and non-proportional allocation (RNA) of particles. In the former, each PE generates particles proportional to the total weight of its particle subset and performs a particle exchange algorithm for load balancing. The number of particles generated by PEs and the transmission of the particles are controlled by a master PE. There is no difference between sequential resampling and RPA in terms of particles generated. However, RPA requires a non-deterministic and complicated interaction between the PEs, which negatively affects the scalability of the algorithm. The second algorithm (RNA) restricts the interaction between the PEs using grouping and local exchange strategies. In the grouping strategy, the PEs form disjoint groups, and each PE is only allowed to communicate the other PEs in its group. In this case, the number of particles in a group stays the same after resampling. Different groups are formed in different sampling instants in order to propagate the large weights to all of the PEs. The groups can also be formed adaptively so that the groups will have similar weights. In the local exchange strategy, the PEs resample their local particles, and they exchange a fixed number of particles to decrease the weight variance among the PEs.

A study focusing on particle exchange is presented in [19]. In this study, new particles are generated by a parallel resampling algorithm that is similar to RPA algorithm in [18], the number of particles produced by each PE is proportional to the total weight of its particle subset. However, here, a PE sends the surplus particles only if the computational burden incurred by the surplus particles exceeds the transmission cost. The surplus particles to be sent are distributed by a dynamic scheduling algorithm in such a way that the PEs having more computing power will have more particles.

A modified version of RNA with the local exchange is proposed in [20]. In this version, the particles to be transmitted are not chosen randomly. Instead, the particles having the largest weights are exchanged before resampling. The work in [20] also shows that these particles best represent the particle subset in terms of Kullback-Leibler divergence, and therefore the exchange of these particles results in more diverse particle subsets. A similar approach is also proposed in [21]. Unlike [20], here, the received

particles are appended to the local particle subset instead of replacing the particles which are sent. The approach in [21] also allows different particle exchange topologies. However, we should note that the particle exchange scheme without replacement may cause distortion in the estimates.

A recent work [22] provides a parallel particle filtering library containing implementations of RNA and RPA algorithms. The library uses a hybrid parallelization model which combines Message Passing Interface (MPI) [23] and Java threads. In addition, the library has three different particle exchange schemes for RPA algorithm: greedy, sorted greedy, and largest gradient. The particle exchange schemes match the PEs having surplus of particles with the PEs having shortage of particles such that the communication overhead induced by the particle exchange is reduced.

A different approach [24] utilizes Independent Metropolis Hastings (IMH) method to perform resampling. Each PE generates a Markov chain of local particles using a uniform distribution as the proposal. A candidate particle is accepted or rejected according to the ratio between the weight of the current particle and the weight of the candidate particle. After sufficiently large number of particles are generated, the initial particles are discarded as a part of the Markov chain burn-in period, and the remaining particles compose the new resampled particle subset. One obvious disadvantage of this method is that one particle subset eventually dominates the others by holding all the weights as in [16]. Besides, the number of iterations needed to make the bias insignificant could be very large especially when the weight variance among the particle subsets is large.

Another different method with provable convergence is suggested in [25]. This method performs resampling in two hierarchical steps. First, the particle subsets are resampled according to their weights as if they are single particles (islands). Second, the particles within the subsets are resampled locally by different PEs. The bottleneck of this method in a distributed environment is to replicate and transmit the whole particle subsets after the first step. The variants of this method, ϵ-bootstrap and Effective Sample Size (ESS) interaction schemes, are described in the same article. In ϵ-

bootstrap interaction scheme, a particle or a subset is not resampled with a probability proportional to its weight. In ESS interaction scheme, resampling is skipped if the ESS of local particles or subsets is higher than a particular threshold. The variance and bias of the estimates calculated by these methods are also studied and compared in [25].

In [26], SMC methods with the constrained interaction are generalized and theoretically studied. The interaction between particles during resampling is represented by a Markov transition matrix α. In this scheme, using a sparse matrix can significantly reduce the interaction between particles thereby reducing the communication cost. However, the weight variance of the resultant particles could be very large. The work in [26] shows that the stability of SMC methods can be guaranteed by ensuring a few conditions controlling ESS. Motivated by this, a parallel implementation of resampling [27], forest resampling, satisfying the required conditions is introduced. Forest resampling operates on disjoint groups of particle subsets forming the subtrees of a logical tree topology of a distributed computer architecture. The subtrees are selected such that the tradeoff between the degree of interaction and the weight variance is balanced.

2.2. GPU Implementation of Resampling

The GPU implementation of SMC methods has also been studied in the literature. An early study [28] introduces the GPU implementations of three resampling algorithms: multinomial, systematic and minimum error resampling. The resampling algorithms are described as the combination of core primitives of parallel computing: scan, reduce and sort. The multinomial and minimum error resampling algorithms use the parallel sort algorithm and therefore they have $\mathcal{O}(log(N)^2)$ time complexity instead of $\mathcal{O}(log(N))$. After the calculation of the offspring numbers using the resampling methods above, the offspring particles are produced and redistributed according to a divide-and-conquer approach.

Another study in [29] proposes a general particle filter implementation developed by using general-purpose computing techniques on GPUs. In [29], the cumulative sum

of the weights are calculated by two-level upsweep/downsweep method [30], then a parallel stratified or systematic resampling calculates the offspring numbers of particles. After that, the offspring particles are produced and distributed using the rasterization process, which is a graphics-specific feature of GPUs.

With the introduction of CUDA [31], the general-purpose computing on GPUs have become more popular. CUDA is parallel computing platform for Single Program Multiple Data (SPMD) parallel programming style. In CUDA, a kernel function is executed by threads generated by CUDA runtime system. The threads are grouped in blocks, and each block is run by only one streaming multiprocessor (SM). Threads in a block can be cheaply synchronized, and they can access a shared memory which is much faster than the global DRAM. A study [32] provides a localized resampling technique which aims to reduce the the global memory accesses. The particle subsets are assigned to thread blocks instead of PEs, and each block performs resampling locally. This method suffers from the same problem as the method in [16], that is the weights of the particles in all blocks but one converge to zero. Finite-redraw importance-maximizing (FRIM) method is proposed to alleviate this problem. In this prior editing technique, each particle is redrawn many times and the particle that maximizes the likelihood is chosen, which results in more balanced weights. However, this only slows down the degeneration of the weights, and the prior editing technique may distort the estimates substantially.

Another CUDA implementation of resampling, Shared-Memory Systematic Resampling (SMSR), is presented in [33]. In SMSR, first the cumulative sum of the particle weights is calculated using upsweep/downsweep method [30]. After that, each thread computes the offspring number of a single particle, and replicates the particle accordingly. The main drawback of SMSR is the thread divergence during replication of the particles. For example, if the offspring number of a particular particle is much larger than the others, the replication of the particles is mostly done by a single thread.

A new resampling algorithm, Metropolis resampling, which is suitable for GPU architectures is proposed in [34]. In Metropolis resampling, each thread runs an inde-

pendent Metropolis method where the stationary distribution is a categorical distribution over particles with their associated weights. As the number of iterations gets larger and larger, Metropolis resampling converges to multinomial resampling. One disadvantage of this method is that if the variance among the particle weights is high, the number of iterations required for convergence may become very large. Metropolis resampling is studied and extended in [35]. In addition, the work in [35] proposes an alternative algorithm: rejection resampling. In rejection resampling, each thread uniformly draws a random particle and a random number on the interval $[0, 1]$ repeatedly until the random number is greater than the particle weight divided by the largest weight. The weak point of rejection resampling is that the number of iterations executed by each thread may not be the same. The resulting thread divergence could be a serious issue in the GPU context.

3. PARALLEL COMPUTING ARCHITECTURES

The clock rate of the processors have remained nearly constant throughout the last decade because of the overheating problem. Furthermore, the architectural improvements involving bit level parallelism, pipelining, multiple functional units and larger cache size have reached their limits. As a result, the annual increase in the performance of processors dropped from 50% to 22% approximately [36]. The demand for increased speed, on the other hand, keeps growing, and parallel computing has become the prominent way to meet it.

Parallel computing has been extensively used in high performance computing (HPC) for decades. Today, HPC clusters are even available in the cloud, on demand. Examples include Amazon Elastic Compute Cloud [37] and Microsoft Azure [38]. Moreover, thanks to recent advancements, multicore processors and many core GPUs have become available in standard desktops, which makes parallel computing architectures accessible to a broad community [36]. However, designing generic parallel algorithms that can efficiently run on these architectures is not a straightforward task since these architectures have different design philosophies. The performance of a parallel program strongly depends on how the program is mapped the resources of the architecture. Here we discuss the important aspects of cluster systems, shared memory architectures and general purpose GPUs.

3.1. Cluster Systems

Cluster systems consist of independent nodes connected by an interconnection network. The architecture of clusters is based on Multiple Instruction, Multiple Data (MIMD) model according to Flynn's taxonomy. In MIMD model, each PE has a private program memory i.e. each PE runs its own instruction set independently and asynchronously. Besides PEs could load and process different pieces of data, and they are capable of writing values to the data memory.

Flynn's taxonomy is inadequate to capture the characteristics of cluster systems. Their distributed memory organization distinguishes cluster systems from other MIMD architectures. A classification based on memory organization is shown in Figure 3.1. Each node in a cluster has an independent processor having a private memory that cannot be accessed by other nodes. The only way to exchange information between cooperating nodes including synchronization is to perform message-passing communications [36].

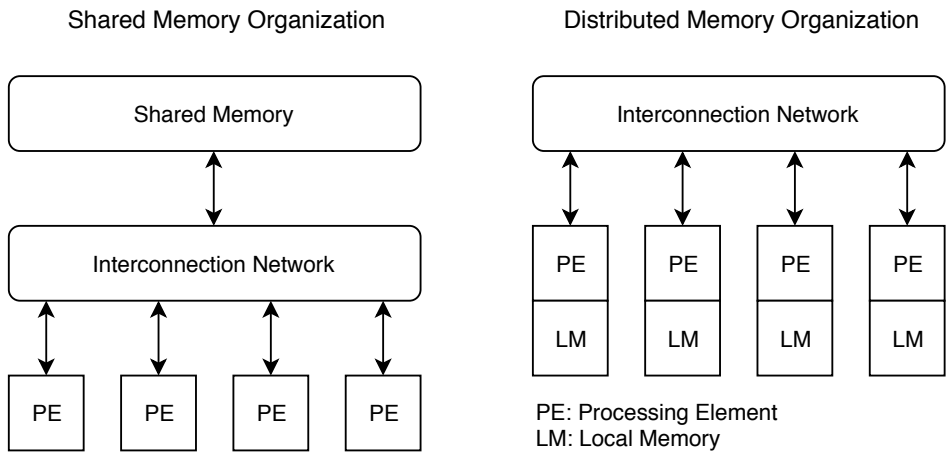

Figure 3.1. Shared and Distributed Memory Organization of MIMD Architectures [36].

Communicating a message in a cluster system is a lot slower than a local memory access, and it can be affected by a number of factors, such as network topology, routing algorithm and switching strategy [36]. Therefore, a computational model is needed to develop and evaluate parallel algorithms. The model should hide the unnecessary details of specific systems, but at the same time it should be expressive enough to reflect important aspects of clusters. Parallel random-access machine (PRAM), bulk synchronous parallel (BSP) and LogP [39] models are popular examples for computational models.

We use LogP as the computational model for cluster systems, because it provides more general parameters capable of representing a broad range of parallel machines. LogP model characterizes distributed MIMD based computing systems by four param-

eters:

- L (Latency): an upper bound on the latency incurred in transmitting a message from a source PE to its target PE.
- o (overhead): the time required by a PE to perform the transmission or reception of a message. A PE cannot execute any other operation during this time.
- g (gap): minimum possible time interval between consecutive transmission or reception operations at a PE.
- P (Processors): the number of processing units.

Moreover, this model assumes that the network has a finite capacity, which is to say that at most $\lceil L/g \rceil$ messages can be communicated at any time. L, o and g reflects the important bottlenecks of cluster systems, and must be considered carefully in designing parallel algorithms.

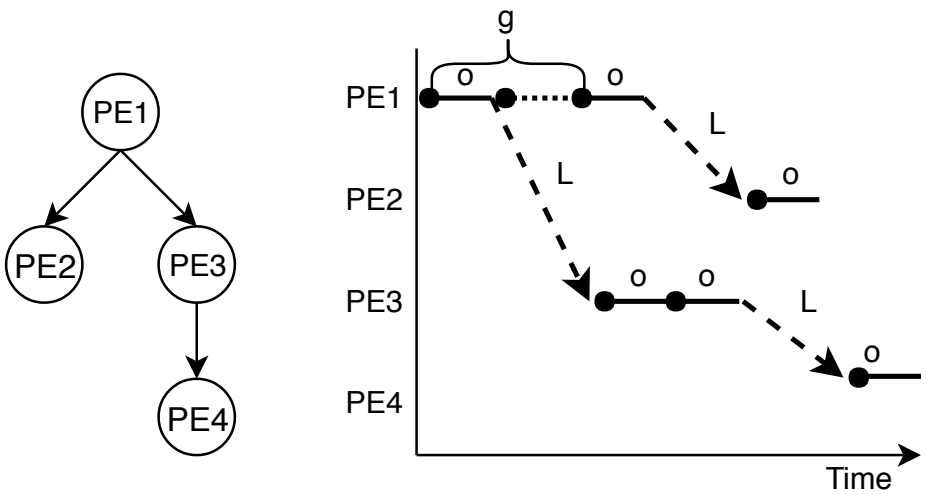

Figure 3.2. An example of broadcasting algorithm based on the optimal broadcasting tree in [40].

The communication tree and the timing diagram of a broadcasting algorithm is shown in Figure 3.2. In this algorithm, PE1 does not send the data to all the other PEs, because the overhead of successive transmission operations and the gap required between them dominates the network latency. Instead, PE1 broadcasts the data to

PE2 and PE3, and PE3 sends the data to PE4 right after it receives. By means of this, the total execution time is reduced.

3.2. Shared Memory Architectures

The number of transistors on an integrated circuit doubles approximately every two years, according to Moore's law. The increase in the number of transistors has been used to increase the number of cores in a processor for the last decade. Today, quad-core processors are the standard for desktop computers, and octa-core even 16-core processors are available off the shelf [36]. Similar to cluster systems, each core in a multicore processor could execute its separate instructions on different data elements. However the cores do not communicate explicitly via message passing, instead, cores exchange information by writing and reading a shared memory.

Accessing a global shared memory is a lot faster than communicating a message through a network. The price to pay for this advantage is that the collisions incurred by concurrent accesses may result in unpredictable race conditions. To avoid race conditions, the cores must be synchronized, which could also lead an overhead. The goal in designing efficient parallel programs for shared memory architectures such as multicore processors is to achieve a good load balance between synchronization points while keeping the number of synchronizations as small as possible.

3.3. General Purpose GPUs

Graphics processing units (GPUs) are another type of shared memory architectures, however the design philosophy of GPUs is fundamentally different from that of multicore processors. Their original purpose is to accelerate graphics operations, which are inherently massively parallelizable, and therefore they have been designed to optimize execution throughput of parallel algorithms. The throughput oriented design of GPUs saves the the chip area to put more compute cores in exchange for long latency [41].

The performance of GPUs is much greater than CPUs in terms of execution throughput. This performance gap has attracted considerable interest from both research and developer communities. At the beginning, a parallel algorithm must be implemented using graphics operations provided by a high level library such as DirectX or OpenGL to leverage GPUs. Therefore general purpose computing on GPUs (GPGPU) was limited and very difficult to use at that time. Later Nvidia realized the growing demand for GPGPU and released CUDA (Compute Unified Device Architecture) to meet it [41].

A CUDA program runs on a heterogeneous computing platform with a CPU (host) and one or more GPUs (devices). The execution of a CUDA program starts with the execution of the host program. The host program can invoke data-parallel device programs by launching kernels. A kernel function generates a number of threads, called a grid, which are organized and mapped to the compute cores on the device. A typical CUDA-capable device consists of independent streaming multiprocessors (SMs) that are connected to a global high bandwidth memory. Each SM has a number of CUDA cores, also called streaming processors. A grid of threads are divided into thread blocks and each block are assigned to a SM in arbitrary order [41]. An example is shown in Figure 3.3.

Thread blocks are partitioned into 32-thread warps, after they are assigned to a SM [41]. The SM schedules the warps instead of individual threads, and the warps are executed by a Single Instruction, Multiple Data (SIMD) hardware. That is, each thread in a warp applies the same instruction on its private data element. If the threads in a warp have different execution paths (e.g. if-else statements), the SIMD hardware passes through all the divergent paths sequentially thereby increasing the execution time.

Global memory access is another critical factor that can affect the performance of a CUDA program. The global memory is based on DRAM technology having long access latencies. Accessing the global memory frequently results in a considerable performance degradation, which could make the program impractical. The global memory

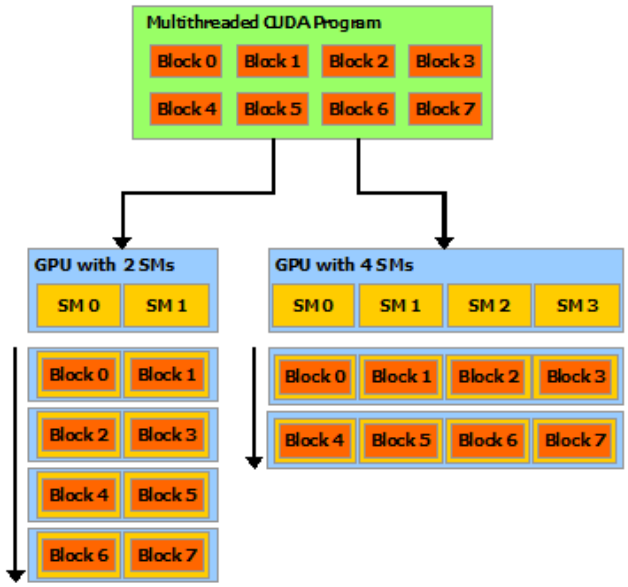

Figure 3.3. Automatic Scalability in CUDA, image taken from [31].

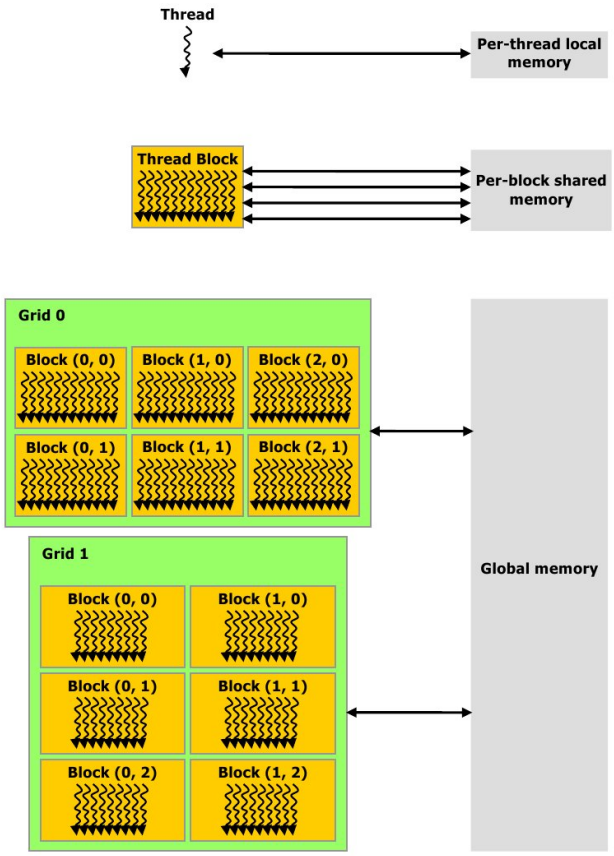

Figure 3.4. CUDA Memory Hierarchy, image taken from [31].

traffic can be reduced by utilizing the shared memory, another programmable memory type provided by CUDA. The size of the shared memory is limited, but it is extremely fast compared to the global memory. During the execution, the threads in a block can access a shared memory allocated for the block, and thus they can share information and store intermediate values at a low cost. The memory hierarchy of CUDA is shown in Figure 3.4. Another important technique to increase the efficiency of global memory operations is memory coalescing. The hardware is able to combine the load or store operations issued by the threads in a warp when the corresponding memory locations are consecutive. Therefore the data access patterns of a CUDA program should be designed to enable memory coalescing in order to achieve a high data access rate.

4. SEQUENTIAL MONTE CARLO

Sequential Monte Carlo (SMC) methods are a set of Monte Carlo techniques for sampling from a high dimensional target distribution $\pi(x_{1:t})$ defined on a product space $\mathcal{X}^n$. The key idea in SMC is to combine Sequential Importance Sampling (SIS) with resampling.

4.1. Sequential Importance Sampling

Sequential Importance Sampling (SIS) is an instance of Importance Sampling (IS) which allows to sample from high dimensional distributions in an efficient way. In SIS, instead of sampling directly from the target distribution $\pi(x_{1:t})$, the samples are sequentially drawn from a sequence of simpler proposal distributions. Consider the proposal distribution $q(x_{1:t})$ which is factorized as

$$q(x_{1:t}) = q(x_1) \prod_{k=2}^{t} q(x_k|x_{1:k-1}). \tag{4.1}$$

SIS constructs the samples $X_{1:t}^i$ recursively. At the initial step, SIS samples $X_1^i \sim q(x_1)$, then for $k = 2, \ldots, t$ SIS samples $X_k^i \sim q(x_k|X_{1:k-1}^i)$ using the previous samples $X_{1:k-1}^i := \{X_1^i, \ldots, X_{k-1}^i\}$. SIS assigns proper weights (importance weights) to the samples, and updates the weights at each time step to correct the discrepancy between the target distribution $\pi(x_{1:t})$ and the proposal distribution $q(x_{1:t})$.

SIS works even when the normalization constant of $\pi(x_{1:t})$ is intractable. Suppose we are able to compute $\phi(x_{1:t}) : \mathcal{X}^t \to \mathbb{R}^+$ where,

$$\pi(x_{1:t}) = \frac{\phi(x_{1:t})}{Z_t} \tag{4.2}$$

$$Z_t = \int \phi(x_{1:t}) dx_{1:t}. \tag{4.3}$$

Here Z_t denotes the intractable normalization constant of $\phi(x_{1:t})$. In this case, the weight updates to be performed by SIS can be formalized as follows:

$$\begin{aligned} W_t(x_{1:t}) &= \frac{\phi(x_{1:t})}{q(x_{1:t})} \\ &= \frac{\phi(x_{1:t-1})}{q(x_{1:t-1})} \frac{\phi(x_{1:t})}{\phi(x_{1:t-1})q(x_t|x_{1:t-1})} \\ &= W_{t-1}(x_{1:t-1}) \frac{\phi(x_{1:t})}{\phi(x_{1:t-1})q(x_t|x_{1:t-1})}. \end{aligned} \tag{4.4}$$

Using Equation 4.1 and 4.4, SIS algorithm can be easily constructed. We summarize SIS algorithm in Figure 4.1. Here N denotes the number of samples.

for $i = 1$ to N **do**
  $X_1^i \sim q(x_1)$
  $W_1^i \Leftarrow \frac{\phi(X_1^i)}{q(X_1^i)}$
end for
for $t = 2, 3, \ldots,$ **do**
  for $i = 1$ to N **do**
    $X_t^i \sim q(x_t|X_{1:t-1}^i)$
    $W_t^i \Leftarrow \frac{\phi(X_{1:t}^i)}{\phi(X_{1:t-1}^i)q(X_t^i|X_{1:t-1}^i)} W_{t-1}^i$
  end for
end for

Figure 4.1. Sequential Importance Sampling

We can calculate the Monte Carlo approximation of $\pi(x_{1:t})$ using the samples and the weights for each time step t by the following equation:

$$\hat{\pi}(x_{1:t}) = \sum_{i=1}^{N} w_t^i \delta_{X_{1:t}^i}(x_{1:t}). \tag{4.5}$$

where w_t^i denotes the normalized weight of the ith sample.

$$w_t^i = \frac{W_t^i}{\sum_{j=1}^{N} W_t^j} \tag{4.6}$$

The SIS approximation of the expectation of a given test function of interest $\varphi(x_{1:t}) : \mathcal{X}^t \to \mathbb{R}$ can be computed by

$$I_{\mathrm{SIS}}(\varphi(x_{1:t})) = \sum_{i=1}^{N} w_t^i \varphi(X_{1:t}^i). \tag{4.7}$$

We can also obtain an unbiased importance sampling estimate of the normalization constant by

$$\hat{Z}_t = \frac{1}{N} \sum_{i=1}^{N} W_t^i. \tag{4.8}$$

Both $I_{\mathrm{SIS}}(\varphi(x_{1:t}))$ and $\hat{Z}_t$ approximations satisfy a Central Limit Theorem (CLT) i.e. the asymptotic variances of these approximations are of order $\mathcal{O}(1/N)$ [5]. Although SIS allows us to sample from the target distribution $\pi(x_{1:t})$ in an efficient way, it has a severe drawback. The variance of the SIS estimates increases, typically exponentially, as t increases, which makes SIS impracticable.

4.2. Resampling and Sequential Monte Carlo

The estimates provided by SIS algorithm are not reliable for large t. In practice after a few time steps, most of the weights of the samples are very close to zero and they require many computational effort although they have little effect on the result. This well-known phenomenon is called the weight degeneracy.

Resampling is an intuitive idea that partially addresses the weight degeneracy. Resampling eliminates the samples with small weights and increases the number of samples with large weights. There are many ways to perform resampling including

popular systematic, residual and multinomial Resampling [42]. The goal is to obtain equally-weighted samples approximating to $\pi(x_{1:t})$ using the SIS samples. For example, let the probability that the sample $X^j_{0:t}$ at the index j becomes the ancestor of the sample $\bar{X}^i_{0:t}$ at the index i after resampling be expressed by

$$\begin{aligned} p(\bar{X}^i_{0:t} = X^j_{0:t}) &= \frac{W^j_t}{\sum_{k=1}^N W^k_t} \\ &= w^j_t \end{aligned} \tag{4.9}$$

Performing resampling using the probabilities in Equation 4.9 is equivalent to sampling from a multinomial distribution if the new samples are conditionally independent. Let N^i_t denote the offspring number associated with ith sample. The new equally-weighted samples are drawn from a multinomial distribution: $N^{1:N}_t \sim \text{Mult}(N, w^{1:N}_t)$ where $N^{1:N}_t := \{N^1_t, N^2_t, \ldots, N^N_t\}$.

We can now construct a generic SMC method using SIS and resampling. We add an extra step to SIS algorithm in Figure 4.1.

for $i = 1$ to N **do**
  $X^i_1 \sim q(x_1)$
  $W^i_1 \Leftarrow \dfrac{\phi(X^i_1)}{q(X^i_1)}$
end for
for $t = 2, 3, \ldots,$ **do**
  $A^{1:N}_{t-1} \sim \mathcal{F}(a^{1:N}_{t-1} | W^{1:N}_{t-1})$
  for $i = 1$ to N **do**
    $X^i_t \sim q(x_t | X^{A^i_{t-1}}_{1:t-1})$
    $W^i_t \Leftarrow \dfrac{\phi(X^i_{1:t})}{\phi(X^{A^i_{t-1}}_{1:t-1}) q(X^i_t | X^{A^i_{t-1}}_{1:t-1})} \sum_{j=1}^N \dfrac{W^j_{t-1}}{N}$
  end for
end for

Figure 4.2. Sequential Monte Carlo

In Figure 4.2, the ancestor indices $A_{t-1}^{1:N} := \{A_{t-1}^1, \ldots, A_{t-1}^N\}$ are sampled from $\mathcal{F}(a_{t-1}|W_{t-1}^{1:N})$, which is the discrete probability distribution corresponding to the resampling scheme. Here A_{t-1}^i denotes the index of the ancestor of X_t^i.

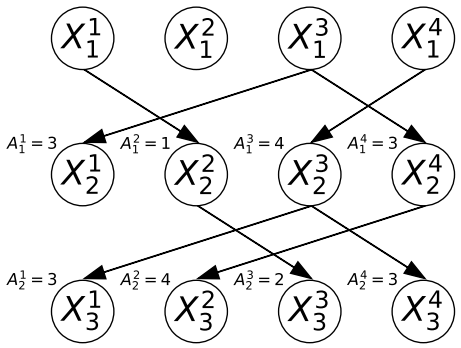

Figure 4.3. Ancestral Lineage Example

Figure 4.3 shows a synthetically generated example of ancestor lineages for 4 samples and 3 steps. For example, the ancestor index A_2^2 of the sample X_3^2 is 4, and $X_{1:3}^2$ is $\{X_1^3, X_2^4, X_3^2\}$.

4.3. Particle Filtering

One of the most successful applications of SMC methods is filtering in state-space models (SSMs). SSMs consists of a state process and an observation process. Consider a Markov state process $\{X_t; t \geq 1\}$ having the following form:

$$X_1 \sim \mu(x_1) \tag{4.10}$$

$$X_t|(X_{t-1} = x_{t-1}) \sim f(x_t|x_{t-1}) \tag{4.11}$$

where $\mu(x_1)$ is the initial density and $f(x_t|x_{t-1})$ is the transition density. The state process is observed through another process $\{Y_t; t \geq 1\}$. In SSMs, the observations are independent given the states, and distributed by a likelihood density given by

$$Y_t|(X_t = x_t) \sim g(y_t|x_t). \tag{4.12}$$

The inference goals in SSMs involve filtering, marginal likelihood computation and smoothing. The problem of filtering is to calculate $p(x_t|y_{1:t})$, the distribution of the present state given all the available observations. The filtering distribution $p(x_t|y_{1:t})$ is the marginal of the joint distribution $p(x_{1:t}|y_{1:t})$. Assume we choose $\pi(x_{1:t}) = p(x_{1:t}|y_{1:t})$, and $\phi(x_{1:t}) = p(x_{1:t}, y_{1:t})$ in Equation 4.2, then the normalization constant becomes $Z_t = p(y_{1:t})$. By using

$$\begin{aligned} \frac{\phi(x_{1:t})}{\phi(x_{1:t-1})} &= \frac{p(x_{1:t}, y_{1:t})}{p(x_{1:t-1}, y_{1:t-1})} \\ &= f(x_t|x_{t-1})g(y_t|x_t) \end{aligned} \tag{4.13}$$

we can simplify SMC algorithm in Figure 4.2 into a generic particle filter shown in Figure 4.4.

for $i = 1$ to N **do**
  $X_1^i \sim q(x_1)$
  $W_1^i \Leftarrow \dfrac{\mu(X_1^i)g(y_1|X_1^i)}{q(X_1^i)}$
end for
for $t = 2, 3, \ldots,$ **do**
  $A_{t-1}^{1:N} \sim \mathcal{F}(a_{t-1}^{1:N}|W_{t-1}^{1:N})$
  for $i = 1$ to N **do**
    $X_t^i \sim q(x_t|X_{1:t-1}^{A_{t-1}^i})$
    $W_t^i \Leftarrow \dfrac{f(X_t^i|X_{t-1}^{A_{t-1}^i})g(y_t|X_t^i)}{q(X_t^i|X_{1:t-1}^{A_{t-1}^i})} \sum_{j=1}^{N} \dfrac{W_{t-1}^j}{N}$
  end for
end for

Figure 4.4. Particle Filter

The proposal distribution $q(x_t|x_{1:t-1})$ in Figure 4.4 is a design parameter. Though the optimal choice for proposal distribution is $q(x_t|x_{1:t-1}) = p(x_t|x_{t-1}, y_t)$, it is not always possible to sample from $p(x_t|x_{t-1}, y_t)$ [5]. Another popular choice is using the transition density $q(x_t|x_{1:t-1}) = p(x_t|x_{t-1})$ as in the standard bootstrap filter [3, 43]. In that case, the initial weights become $W_t^i = 1$, and the weights can be updated using

just the likelihood density $W_t^i \Leftarrow g(y_t|X_t^i) \sum_{j=1}^{N} \frac{W_{t-1}^i}{N}$.

The sample paths $\{X_{1:t}^1, \ldots, X_{1:t}^N\}$ provided by the particle filter in Figure 4.4 could be used to approximate $p(x_{1:t}|y_{1:t})$, and thereby its marginals $p(x_k|y_{1:t})$ for $k = 1, \ldots, t$ (see Equation 4.5). However, the number of unique samples approximating $p(x_k|y_{1:t})$ decreases very fast (exponentially) as the difference $t - k$ gets larger and larger. One simple solution to this sample degeneracy is the fixed-lag approximation relying on

$$p(x_k|y_{1:t}) \approx p(x_k|y_{1:k+d}). \tag{4.14}$$

However, the approximation in Equation 4.14 also has its limitations. We do not here include other particle smoothing methods, please refer to [5] for details.

5. PARALLELIZATION OF RESAMPLING

All of the steps of an SMC method except for resampling are readily parallelizable as they do not require sample interaction. Resampling gathers the weights of the samples, replicates the samples according to their weights, and redistributes the resulting samples. Thus, it is not trivial to design a parallel resampling algorithm. In this chapter, we first describe the standard way to parallelize resampling algorithms by taking into account the performance considerations of parallel computing architectures discussed in Chapter 3. Then we provide mathematical formulations for the execution time of the standard method on cluster systems and GPUs.

5.1. Ancestor Indices and Offspring Numbers

A standard resampling algorithm provides equally-weighted samples approximating to $\pi(x_{1:t})$ by simply sampling from the IS approximation $\hat{\pi}(x_{1:t})$ in Equation 4.5. Resampling can be performed by drawing ancestor indices from a categorical distribution with associated weights at time $t+1$

$$A_t^i \sim \mathrm{Cat}([N], w_t^{1:N}), \text{ for all } i \in [N]. \tag{5.1}$$

We denote by $[N] := \{1, 2, \ldots, N\}$ and $w_t^{1:N} := \{w_t^1, w_t^2, \ldots, w_t^N\}$. The conditional independence between the ancestor indices $A_t^{1:N} = \{A_t^1, A_t^2, \ldots, A_t^N\}$ can be identified by a bipartite directed graph. Let $\{A_{t,0}^i\}_{i\in[N]}$ and $\{A_{t,1}^i\}_{i\in[N]}$ denote the values in the vertices of the disjoint subsets respectively, where $A_{t,0}^i = i$ and $A_{t,1}^i = A_t^i$. There is an edge from $A_{t,0}^j$ to $A_{t,1}^i$ if and only if the probability $p(A_t^i = j)$ is greater than zero.

Figure 5.1 shows the conditional independence graph of a standard resampling algorithm. As can be seen, the ancestor index of an offspring sample could be any index. In the first part (a) of Figure 5.2 a random realization of ancestor indices is shown. Since the order of the samples is not important, the ancestor indices can be grouped according to their values as in the part (b) in Figure 5.2. This approach is

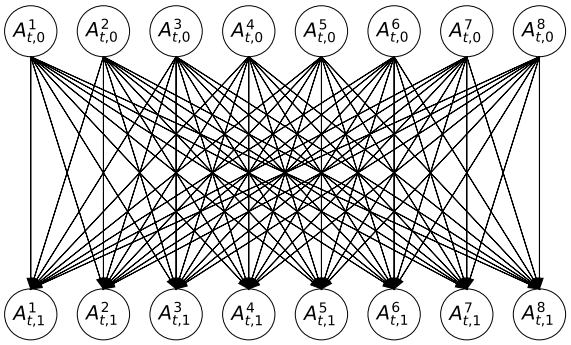

Figure 5.1. Conditional Independecy Graph of Standard Resampling.

identical to drawing offspring numbers (replication numbers) $N_t^{1:N}$ from a multinomial distribution: $N_t^{1:N} \sim \text{Mult}(N, w_t^{1:N})$. Multinomial resampling provides an unbiased approximation for $\hat{\pi}(x_{1:t})$ where $\mathbb{E}[N^i | w_t^{1:N}] = Nw^i$ for all $i \in [N]$. There exist other unbiased resampling methods having smaller variance such as residual and systematic resampling [42], but for the sake of simplicity we consider only multinomial resampling throughout the thesis.

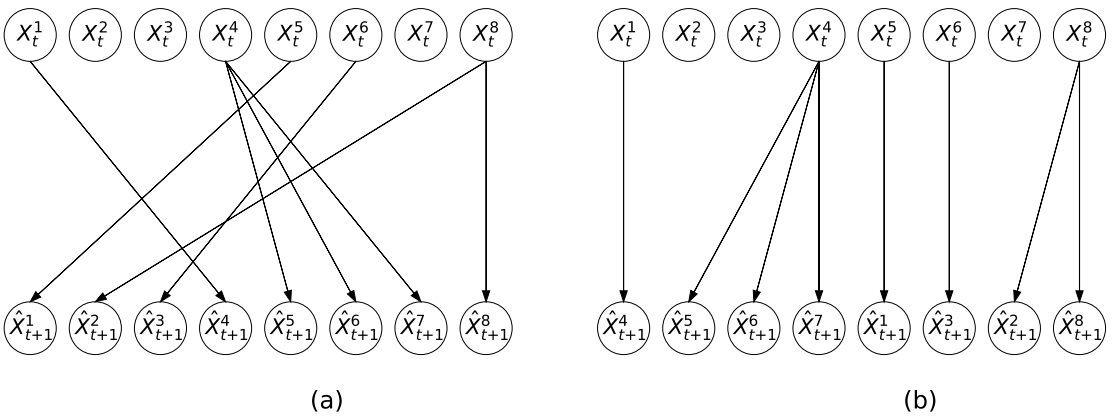

Figure 5.2. An Example of Multinomial Resampling.

5.2. A Parallel Two-Step Resampling Algorithm

Let us assume that the samples are distributed to PEs such that each PE has $M = N/P$ samples. In a typical parallelization approach, resampling is performed in two steps. In the first step, each PE calculates the sum of local weights in parallel

$$\mathbf{W}_t^k = \sum_{i=1}^{M} W_t^{i,k}. \tag{5.2}$$

Here k is the index of the PE and i is the local sample index. After that, the weight sums, which we call the global weights, are normalized $\mathbf{w}_t^k = \dfrac{\mathbf{W}_t^k}{\sum_{l=1}^{P} \mathbf{W}_t^l}$, and the offspring numbers associated to PEs $M_t^{1:P} := \{M_t^1, M_t^2, \ldots, M_t^P\}$ are sampled multinomially $M_t^{1:P} \sim \text{Mult}(N, \mathbf{w}_t^{1:P})$ where $\mathbf{w}_t^{1:P} := \{\mathbf{w}_t^1, \mathbf{w}_t^2, \ldots, \mathbf{w}_t^P\}$. In the second step, each PE independently performs a local resampling algorithm such that the PE k generates M_t^k samples. Then the PEs exchange samples via message-passing communication or a global memory to achieve the load balance. The two-step parallel resampling algorithm is outlined in Figure 5.3.

for $k = 1$ to P **do in parallel**
  $\mathbf{W}_t^k \Leftarrow \sum_{i=1}^{M} W_t^{i,k}$
end for
$\mathbf{w}_t^{1:P} \Leftarrow \text{Normalize}(\mathbf{W}_t^{1:P})$
$M_t^{1:P} \sim \text{Mult}(N, \mathbf{w}_t^{1:P})$
for $k = 1$ to P **do in parallel**
  $A_t^{1:M_t^k,k} \sim \mathcal{F}(a_t^{1:M_t^k,k} | W_t^{1:M,k})$
end for
$\text{DistributeSamples}(A_t^{1:M_t^1,1}, A_t^{1:M_t^2,2}, \ldots, A_t^{1:M_t^P,P})$

Figure 5.3. A Parallel Resampling Algorithm

5.3. Message Passing Implementation

We want to provide some important implementation-specific details before discussing the performance of this algorithm. In our implementation, all the PEs have

the same role, there is no master or slave PE. In a cluster system the global weights $\mathbf{w}_t^{1:P}$ are obtained by all the PEs through a all-to-all broadcast operation. In addition, each PE uses a separate random number generator for global operations, which is initialized with a shared seed. Therefore, PEs are able to compute the same offspring numbers $M_t^{1:P}$ without any communication. Another issue needed to be addressed is redistribution of surplus samples. If the offspring number of a PE M_t^k is greater than M, then the PE k must send $L_t^k = M_t^k - M$ samples to one or more other PEs. If M_t^k is smaller than M, it will receive $|L_t^k|$ samples from other PEs. A matching algorithm is needed to match the PEs having surplus of samples with the PEs having shortage of samples. A good matching should keep the number of communications minimum.

for $k = 1$ to P **do**
  $L_t^k \Leftarrow M_t^k - M$
end for
$s \Leftarrow 1$
$r \Leftarrow 1$
while $s \leq P$ and $r \leq P$ **do**
  if $L_t^s \leq 0$ **then**
    $s \Leftarrow s + 1$
  end if
  if $L_t^r \geq 0$ **then**
    $r \Leftarrow r + 1$
  end if
  if $L_t^s > 0$ and $L_t^r < 0$ **then**
    Schedule($min(L_t^s, |L_t^r|)$)
    $L_t^s \Leftarrow L_t^s - min(L_t^s, |L_t^r|)$
    $L_t^r \Leftarrow L_t^r + min(L_t^s, |L_t^r|)$
  end if
end while

Figure 5.4. A Greedy Matching Algorithm

Here, we describe a simple greedy matching algorithm in Figure 5.4, however other approaches can also be used [22]. The matching algorithm picks a sender s and

a receiver r, and schedules a transmission of $min(L_t^s, |L_t^r|)$ samples. If $L_t^s \geq |L_t^r|$, then another receiver is picked; likewise, if $L_t^s \leq |L_t^r|$, then another sender is picked by the scheduler. The algorithm iterates over the PEs until all the senders and receivers are matched.

The main bottleneck is the transmission the global weights and the surplus samples. The global weights can be broadcasted to all PEs in an efficient way. If PE k sends the global weight to PEs $k+1(\text{mod } P), k+2(\text{mod } P), \ldots, k+P-1(\text{mod } P)$ in that order, then the communication time becomes $L+2o+(P-2)g$ according to LogP model [40]. In transmission of samples, we assume a PE is either a receiver or a sender since having exactly M samples ($M_t^k = M$) for a PE after resampling is unlikely. Let $P_{\max}$ denote one plus the maximum number of transmissions or receptions performed by a PE, then communicating surplus particles takes $L + 2o + (P-2)g$ cycles. In the worst case scenario, a PE receives samples from all of the other PEs. Another important issue is that the offspring numbers sampled using the normalized global weights could be highly unbalanced. In the worst case a PE produces N particles. Now we can provide an expression for the execution time

$$\begin{aligned} T &= T_{\text{weight}} + T_{\text{broadcast}} + T_{\text{ancestor}} + T_{\text{exchange}} \\ &= Mt_w + L + 2o + (P-2)g + \max_{k\in[P]} M_t^k t_a + L + 2o + (P_{\max}-2)g \\ &= \frac{N}{P}t_w + \max_{k\in[P]} M_t^k t_a + 2L + 4o + (P_{\max}+P-2)g. \end{aligned} \tag{5.3}$$

Here T_{weight} is time spent during the calculation of the sum of the local weights, which is fixed for all time instants, T_{ancestor} is the time spent by the PE having the largest offspring number during producing the ancestor indices, and t_w and t_a denote the time required to calculate the sum of two weights and one ancestor index respectively.

5.4. Shared Memory Implementation

In a shared memory architecture, an explicit barrier synchronization is needed before the normalization of the global weights to make sure all the PEs have calculated

the sum of the local weights. Another critical point is that after the offspring numbers are sampled, many PEs may run idle while others generating the ancestor indices, especially when the offspring numbers are highly unbalanced. This problem can be solved easily since PEs can access all of the samples via a global memory. For example, instead of sampling the offspring numbers from a multinomial distribution with the parameter N (number of trials), each PE samples its own offspring numbers $M_t^{1:P,k} \sim \text{Mult}(M, \mathbf{w}_t^{1:P})$ with M trials, then generates the ancestor indices according the $M_t^{1:P,k}$, and this way, the number of ancestor indices to be generated by each PE is fixed and equals M. A better approach exploiting data locality is to perform a matching algorithm as in Figure 5.4 after the offspring numbers are sampled with N trials such that the PEs having shortage of samples generate the surplus ancestor indices of other PEs. The execution time of these approaches can be described as

$$\begin{aligned} T &= T_{\text{weight}} + T_{\text{sync}} + T_{\text{ancestor}} + T_{\text{sync}} \\ &= Mt_w + T_{\text{sync}} + Mt_a + T_{\text{sync}} \\ &= \frac{N}{P}(t_w + t_a) + 2T_{\text{sync}} \end{aligned} \tag{5.4}$$

where T_{sync} is the amount of time required for synchronization.

5.5. GPU Implementation

The GPU implementation of the algorithm in Figure 5.3 is similar to the shared memory implementation because they both utilize a global memory. However, the PEs of GPUs, streaming multiprocessors (SMs), are not sequential computing units, they are based on SIMD model (see Section 3.3). The sum of the local weights $\mathbf{W}_t^k$ are calculated by a thread block having shared resources. There exist a plenty of efficient techniques to be used to calculate the sum, we refer to [41] for details. Once the global weights are calculated, the offspring numbers $M_t^{1:P,k}$ are produced for each block on a host device (CPU). After that, each thread block copies the local weights from the global memory to its shared memory, and calculates the cumulative (prefix) sum of the local weights using work efficient upsweep/downsweep algorithm [30] summarized

in Figure 5.5. Then, each thread in the block independently performs a binary search to calculate an ancestor index.

Although it is not easy to express the execution time of an algorithm running on a GPU in terms of some parameters, the long-latency global memory access is a well known limiting factor for the performance. We can express the execution time of the algorithm in Figure 5.3 as the sum of calculation time of the cumulative sum of the weights T_{cs}, fetching time of the weights from the global memory T_{gmem}, and the time required for a binary search T_{bs} as follows:

$$\begin{aligned} T &= T_{\text{cs}} + T_{\text{sync}} + T_{\text{gmem}} + T_{\text{bs}} + T_{\text{sync}} \\ &= (\log_2 M)t_w + P_{\max}t_{\text{gmem}} + (P_{\max}\log_2 M)t_{bs} + 2T_{\text{sync}} \\ &= \left(\log_2 \frac{N}{P}\right) t_w + P_{\max}t_{\text{gmem}} + \left(P_{\max}\log_2 \frac{N}{P}\right) t_{bs} + 2T_{\text{sync}}. \end{aligned} \tag{5.5}$$

Here, t_{bs} denote time required to perform one step of a binary search, and t_{gmem} is the latency of the global memory. Two synchronization points are needed, one to ensure the cumulative sums are computed, and one to ensure the resampling is completed. We assume the blocks can be executed concurrently. Both t_w and t_{bs} use only the shared memory which is much faster than the global memory. We note that T_{gmem} does not always have to be Pt_{gmem}. In practice, however, the number of global memory accesses is usually of order of P.

for $k = i$ to M **do in parallel**

$W_{cs}^{i} \leftarrow W_{t}^{i,k}$

end for

for $d = 0$ to $\log_2 M - 1$ **do**

for $i = 0$ to $M/2^{d+1} - 1$ **do in parallel**

$W_{cs}^{(i+1)2^{d+1}} \Leftarrow W_{cs}^{(2i+1)2^{d}} + W_{cs}^{(i+1)2^{d+1}}$

end for

end for

$W_{cs}^{M} \leftarrow 0$

for $d = \log_2 M - 1$ down to 0 **do**

for $i = 0$ to $M/2^{d+1} - 1$ **do in parallel**

temp $\Leftarrow W_{cs}^{(2i+1)2^{d}}$

$W_{cs}^{(2i+1)2^{d}} \Leftarrow W_{cs}^{(i+1)2^{d+1}}$

$W_{cs}^{(i+1)2^{d+1}} \Leftarrow$ temp $+ W_{cs}^{(i+1)2^{d+1}}$

end for

end for

Figure 5.5. A Work Efficient Parallel Cumulative Sum Algorithm

6. PARALLEL RESAMPLING WITH BUTTERFLY COMMUNICATIONS

The standard parallelization technique discussed in Chapter 5 provides an efficient resampling algorithm. However, the execution time grows linearly with the number of PEs on computer clusters and GPUs (see Equation 5.3 and 5.5), which is a limiting factor for its scalability especially when the latency of the interconnection network or the latency of the global memory of the GPU is high. In this chapter, we propose a parallel resampling algorithm inspired by Butterfly resampling [14] to solve these performance problems.

6.1. Effective Sample Size and Stability

The algorithm in Figure 5.3 allows any PE to interact with any other PE due to the nature of multinomial distribution. Such a dependency between the PEs is the reason why the execution time grows as the number of PEs increases. One way to mitigate this problem is to put some constraints on the communication between PEs. In such cases, PEs form disjoint groups, and each group independently performs resampling as described in Chapter 5. The weights of the samples in a group become equal after resampling, however the weights of the samples belonging to different groups could be quite different, which may lead to degeneracy of the weights.

The weight degeneracy can be monitored by using effective sample size (ESS) criterion. The ESS at each time instant can be calculated using the weights

$$N_t^{\text{eff}} = \frac{(\sum_{i=1}^{N} \bar{W}_t^i)^2}{\sum_{i=1}^{N} (\bar{W}_t^i)^2}. \tag{6.1}$$

Here the bar in $\bar{W}_t^i$ indicates the weight after resampling. The ESS can take values between 1 and N, indicating the degree of degeneracy. The stability of an SMC method is strongly dependent on the ESS. The study in [26] proposes a generalization of standard

SMC methods, called αSMC, and provides a stability theorem for αSMC stating that a suitable interaction keeping effective sample size (ESS) above a particular threshold is sufficient to guarantee the stability. The αSMC algorithm at each time step picks a matrix α representing the interactions between samples from a set of Markov transition matrices $\mathbb{A}$, and performs resampling accordingly. The denser the matrix α is, the higher ESS to be obtained. The stability theorem ([26], Theorem 2) for αSMC states that, under some regularity conditions on the HMM, there exists a finite constant c such that for any $N \geq 1$,

$$\inf_{t\geq 0} \frac{N_t^{\text{eff}}}{N} \geq \tau \quad \Rightarrow \quad \sup_{t\geq 0} \mathbb{E}\left[\left(\frac{\hat{Z}_t}{Z_t}\right)^2\right]^{1/t} \leq 1 + \frac{c}{N\tau} \tag{6.2}$$

and it follows from Equation 6.2 that,

$$\left.\begin{array}{r} \inf_{t\geq 0} \frac{N_t^{\text{eff}}}{N} \geq \tau, \text{ and} \\ N\tau \geq tc \end{array}\right\} \quad \Rightarrow \quad \mathbb{E}\left[\left(\frac{\hat{Z}_t}{Z_t} - 1\right)^2\right] \leq \frac{2tc}{N\tau}. \tag{6.3}$$

Equation 6.2 and 6.3 show the tradeoff between the stability and the degree of interaction between samples. Increasing the group size i.e. decreasing the number of groups improves the stability of the algorithm, on the other hand the intensive communication within large groups substantially increase the execution time (see Equation 5.3).

6.2. Butterfly Resampling

Butterfly resampling is an instance of augmented resampling, which is a different approach that divides resampling into K steps, where each sample can only interact with a fixed set of other samples. Augmented resampling (AR) imposes constraints on the interactions between samples by using a set of doubly stochastic transition matrices $\{B_1, B_2, \ldots, B_K\}$, each of size $N \times N$ and $(B_K B_{K-1} \ldots B_1)^{i,j} = 1/N$. At step k, the ancestor indices are sampled according to the conditional independence

structure represented by B_k. At the end of the algorithm, equally weighted samples are produced. The pseudocode of AR is given in Figure 6.1. The symbol δ_x denotes the Dirac (unit) delta measure located at x.

for $i = 1$ to N **do**

$A^i_{t,0} \Leftarrow i$

$W^i_{t,0} \Leftarrow W^i_t$

end for

for $k = 1$ to K **do**

for $i = 1$ to N **do**

$W^i_{t,k} \leftarrow \sum_{j=1}^N B^{i,j}_k W^j_{t,k-1}$

$A^i_{t,k} \sim \dfrac{\sum_{j=1}^N B^{i,j}_k W^j_{t,k-1} \delta_{A^j_{t,k-1}}}{W^i_{t,k}}$

end for

end for

for $i = 1$ to N **do**

$A^i_t \Leftarrow A^i_{t,K}$

$\bar{W}^i_t \Leftarrow W^i_{t,K}$

end for

Figure 6.1. Augmented Resampling

There are many ways to choose the interaction matrices, for example when $K = 1$ and $B^{i,j}_1 = 1/N$ for $, i, j \in [N]$, we obtain the standard complete resampling method. In butterfly resampling (BR), the main idea is to break down the complete interaction matrix into well structured sparse matrices so that non-trivial limits for the moments can be established [14]. Let $N = r_1 r_2 \dots r_K$ be the factorization of N, then B_k matrices of BR are formed as follows:

$$B_k := I_{r_K} \otimes \cdots \otimes I_{r_{k+1}} \otimes U_{r_k} \otimes I_{r_{k-1}} \otimes \cdots \otimes I_{r_1}. \tag{6.4}$$

Here the symbol $\otimes$ denotes Kronecker product, I_d denotes the $d \times d$ identity matrix, and U_{r_k} is the $r_k \times r_k$ matrix which has $1/r_k$ as every entry. We can see the matrices satisfy the condition $\prod_k B_k = U_N$ and each matrix A_k ensures that the samples interact

in small groups consisting of r_k particles at step k. For example, assume we have $N = 8$ and $r_1 = r_2 = r_3 = 2$, then the corresponding interaction matrices are

$$B_1 = \frac{1}{2}\begin{bmatrix} 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & 1 \end{bmatrix}, \quad B_2 = \frac{1}{2}\begin{bmatrix} 1 & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ 1 & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & 1 \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & 1 \end{bmatrix},$$

$$B_3 = \frac{1}{2}\begin{bmatrix} 1 & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & 1 \\ 1 & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & 1 \end{bmatrix}. \tag{6.5}$$

The conditional independence graph of BR resampling is depicted in Figure 6.2 (a). As can be seen, the interactions in one step of BR are much more sparse than the interactions in multinomial resampling in Figure 5.1. However, BR has two major disadvantages. First, some extra noise is added in each step of BR. For instance, when N is factorized as $N = r \times r \times r \ldots \times r$, the error associated with BR is of order $\sqrt{\frac{log_r N}{N}}$ which is larger than the standard error of order $1/\sqrt{N}$. The other problem is that after the completion of each step, a barrier synchronization is needed, which significantly degrades the performance. The authors of [14] also propose another BR instance, called mixed BR, in which N is factorized as $N = r \times N/r$, and they show mixed BR has $\sqrt{N}$ scaling. The graph associated with mixed BR is shown in Figure 6.2 (b).

6.3. Butterfly Communications

We propose a new parallel resampling algorithm, which we call resampling with butterfly communications (RBC), inspired by butterfly resampling in [14] and the adaptive strategies in [26]. Our aim is to eliminate the dependency of the performance on the number of PEs, which limits the scalability. Consider the BR instance which factorizes N as $M \times 2 \times 2 \times \ldots \times 2$, and the corresponding interaction matrices $\{B_1, B_2, \ldots, B_{p+1}\}$.

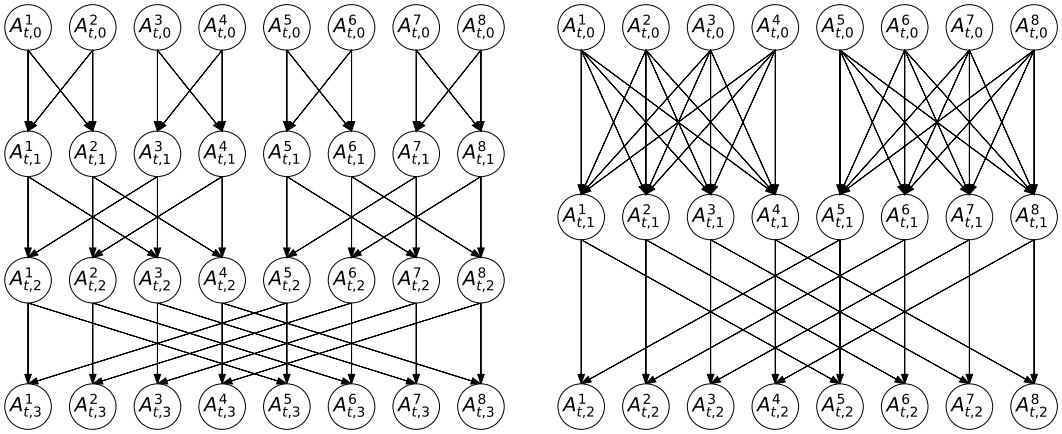

Figure 6.2. Conditional Independence Graph of Butterfly Resampling.

Here the number of steps is $p+1$ where $2^p = P$. We define a new set of matrices $\mathbb{A} = \{C_i \mid C_i = B_{i+1}B_1,\ i \in [p]\}$. For instance if $P = 4$, then C_1 and C_2 become

$$C_1 = \frac{1}{2}\begin{bmatrix} U_M & U_M & \cdot & \cdot \\ U_M & U_M & \cdot & \cdot \\ \cdot & \cdot & U_M & U_M \\ \cdot & \cdot & U_M & U_M \end{bmatrix}, C_2 = \frac{1}{2}\begin{bmatrix} U_M & \cdot & U_M & \cdot \\ \cdot & U_M & \cdot & U_M \\ U_M & \cdot & U_M & \cdot \\ \cdot & U_M & \cdot & U_M \end{bmatrix} \tag{6.6}$$

At each time step t, RBC selects the matrix $\alpha = C_k$ where $k = t \bmod p$, and performs resampling according to the αSMC algorithm. The interaction mechanism specified by C_k is the following: each PE s is paired with $r = p \oplus k$ where the symbol $\oplus$ denotes the bitwise xor-operation (exclusive or operation), and each pair performs complete resampling as in Figure 5.3, and thus the ESS is always kept above $2M$ ($\tau = 2/P$). PEs form different pairs at different time instants according to the butterfly structure outlined in the part (a) in Figure 6.2 (see also [36]). Intuitively, this rapidly propagates the large weights to all of the PEs. We also note that RBC can be seen as an instance of RNA in [18] with a certain re-grouping strategy.

The constrained interactions in RBC lead to a significant speed improvement. Recall from Equation 5.3 that the communication time incurred by inter-message gaps (g) grows linearly with the number of PEs (P) in RPA. On the other hand, in RBC, the PEs are constrained to communicate in pairs, and therefore the gap term in Equation 5.3 is eliminated. The execution time on a computer cluster becomes

$$T^{\mathrm{RBC}} = \frac{N}{P} t_w + \max_{k \in [P]} M_t^k t_a + 2L + 4o. \tag{6.7}$$

Moreover, the upper bound for the maximum number of ancestor indices produced by a PE is $\max_{k \in [P]} M_t^k \leq 2M$, which is substantially smaller than $\max_{k \in [P]} M_t^k \leq N$. Similarly, RBC removes the factor P, the number of PEs, from the global memory access time T_{gmem} and the time spent in binary search T_{bs} in Equation 5.5. Thus the execution time on a GPU is reduced to

$$T^{\mathrm{RBC}} = \left(\log_2 \frac{N}{P} \right) t_w + 2t_{\mathrm{gmem}} + \left(2\log_2 \frac{N}{P} \right) t_{bs} + 2T_{\mathrm{sync}}. \tag{6.8}$$

6.3.1. Number of Communicating Pairs

The latency of the network L with the overhead incurred by communications o in Equation 6.7 could still reduce the performance of the RBC algorithm described above, especially when PEs are interconnected by a network with limited bandwidth and significant latency. We therefore put a limit on the number of communicating pairs during resampling by using more sparse interaction matrices.

Let n denote the number of pairs allowed to communicate. For example, in the standard RBC algorithm described above, $n = P/2$. We compose a new set of interaction matrices $\mathbb{A} = \{C_{i,j} \mid i \in [p],\ j \in [P/2n]\}$. At time step t, the matrix $C_{i,j}$ where $i = \left\lceil \frac{2nt}{P} \right\rceil \bmod p,\ j = t \bmod \frac{P}{2n}$ is selected. The idea here is to separate the communicating pairs at one time instant in the standard RBC into distinct sets, each having n pairs, and to allow the sets of pairs to communicate one at a time. An example of this communication pattern is illustrated in Figure 6.3.

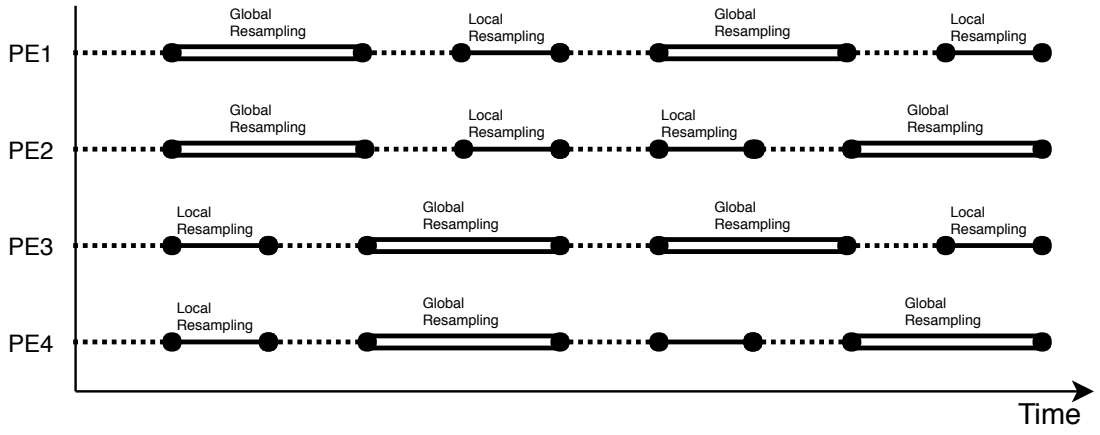

Figure 6.3. An Illustration of Pair Communications.

Suppose, for example, we set $n = 1$, then B_2 is factorized as follows:

$$B_{2,l} := \begin{bmatrix} I_{2M(l-1)} & \cdot & \cdot \\ \cdot & U_2 \otimes I_M & \cdot \\ \cdot & \cdot & I_{N-2Ml} \end{bmatrix}. \tag{6.9}$$

Here we only explain how the matrix B_2 is factorized, but the method can be generalized without difficulty for the rest of the matrices. The corresponding interaction matrices can be easily calculated by $\mathbb{A} = \{C_{i,j} \mid C_{i,j} = B_{i+1,j}B_1,\ i \in [p],\ j \in [P/2n]\}$. For example, if $P = 4$ and $n = 1$, then $C_{1,1}, C_{1,2}, C_{2,1}$ and $C_{2,2}$ are the following matrices:

$$C_{1,1} = \begin{bmatrix} U_M/2 & U_M/2 & \cdot & \cdot \\ U_M/2 & U_M/2 & \cdot & \cdot \\ \cdot & \cdot & I_M & \cdot \\ \cdot & \cdot & \cdot & I_M \end{bmatrix}, \quad C_{1,2} = \begin{bmatrix} I_M & \cdot & \cdot & \cdot \\ \cdot & I_M & \cdot & \cdot \\ \cdot & \cdot & U_M/2 & U_M/2 \\ \cdot & \cdot & U_M/2 & U_M/2 \end{bmatrix},$$

$$C_{2,1} = \begin{bmatrix} U_M/2 & \cdot & U_M/2 & \cdot \\ \cdot & I_M & \cdot & \cdot \\ U_M/2 & \cdot & U_M/2 & \cdot \\ \cdot & \cdot & \cdot & I_M \end{bmatrix}, \quad C_{2,2} = \begin{bmatrix} I_M & \cdot & \cdot & \cdot \\ \cdot & U_M/2 & \cdot & U_M/2 \\ \cdot & \cdot & I_M & \cdot \\ \cdot & U_M/2 & \cdot & U_M/2 \end{bmatrix} \tag{6.10}$$

The RBC algorithm with $n < \frac{P}{2}$ is suitable for pipelining. While some pairs are communicating, others do not have to be idle, they can proceed to the next sampling instant. Each PE has to communicate only once in $\frac{P}{2n}$ time steps. As a result the communication time is reduced by a factor $\frac{2n}{P}$. The execution time in Equation 6.7 can be updated as follows:

$$T^{\text{RBCn}} = \frac{N}{P} t_w + \max_{k \in [P]} M_t^k t_a + (2L + 4o)\frac{2n}{P}. \tag{6.11}$$

We can control the communication time by adjusting the parameter n. However, the minimum possible ESS is decreased to M when $n < P/2$. Besides, partial mixing, especially when n is small, may not be enough to propagate large weights to all of the PEs. Therefore selecting n is a tradeoff between the execution time and the stability of the RBC algorithm.

7. EXPERIMENTS AND RESULTS

In this chapter, we present our simulation results. We conducted our experiments on two parallel computing environments, a cluster of server nodes and a GPU. The cluster consists of nodes having Intel(R) Xeon(R) CPU E5-2650L v3 @ 1.80GHz. We ran the simulations on 16 separate cores each belonging to a different node. The GPU we used in our experiments is Nvidia GeForce GTX 1070 which has CUDA compute capability of 6.1. We used 1024 threads per block, which is the maximum possible number of threads in a block on a device with CUDA compute capability 6.1. Each thread block can perform operations on at most 1024 particles, since the resources that can be allocated for a thread block is limited on a GPU. Therefore the number of blocks used for N particles is $N/1024$.

We implemented the resampling algorithms in Python using the libraries Numba [44] and MPI for Python [45]. Numba allows just-in-time compilation of Python code into CUDA kernels, and supports explicit parallel loops for high performance computing. MPI for Python (mpi4py) provides Python bindings for the Message Passing Interface (MPI) standard. We ran our simulations for 512 time steps with $N = 2^9, 2^{10}, \ldots, 2^{15}$ particles. We recorded the total execution time and the amount of time spent in resampling. Then we calculated the average execution and resampling time for a single time step. We also calculated the effective sample size and the accuracy of the approximations for each time step.

We used three particle filtering applications. The first application is a linear Gaussian state space model (LGSSM) where we can compute the optimal filtering estimates by the Kalman filter [1]. We compare the particle approximations of the RBC and multinomial resampling algorithms with the optimal estimates to evaluate the accuracy. The second application is indoor positioning using bluetooth low energy (BLE) messages. We used the collected data set in [46] consisting of received signal strength indicator (RSSI) values for different locations to construct an observation model. Similarly we evaluate the accuracy of the algorithms by comparing with the

optimal estimates calculated by the HMM filter [2]. In the third application, we used the application of tracking the state of the chaotic Lorenz system, which has highly nonlinear characteristics. We chose an observation model with small variance to observe the effect of weight degeneracy on the performance of the algorithms.

7.1. Linear Gaussian State Space Model

Our first application is the tracking of the position of a target moving within x-y plane according to a simple linear Gaussian state space model (LGSSM). Consider an LGSSM having the following initial, transition and observation processes:

$$\mathbf{x}_0 \sim \mathcal{N}(0, \mathbf{\Sigma}_0) \tag{7.1}$$

$$\mathbf{x}_t|\mathbf{x}_{t-1} \sim \mathcal{N}(\mathbf{F}\mathbf{x}_{t-1}, \mathbf{Q}),\ t \geq 1 \tag{7.2}$$

$$\mathbf{y}_t|\mathbf{x}_t \sim \mathcal{N}(\mathbf{H}\mathbf{x}_t, \mathbf{R}),\ t \geq 1. \tag{7.3}$$

The state vector is four-dimensional $\mathbf{x}_t = (x, v_x, y, v_y)_t^\intercal$ where x and y are coordinates of the target, and v_x and v_y are corresponding velocities. The matrices F and H are the transition and observation matrices respectively

$$\mathbf{F} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}. \tag{7.4}$$

The target starts with a random position and random velocity, and basically performs a simple random walk in which a Gaussian noise is added to its position and velocity at each time step. The velocity of target is hidden and it can be inferred by only using noisy observation of the position. In our implementation all of the covariance matrices

are diagonal matrices:

$$\mathbf{\Sigma_0} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix}, \quad \mathbf{Q} = \begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{R} = \begin{bmatrix} 4 & 0 \\ 0 & 4 \end{bmatrix}. \tag{7.5}$$

An example trajectory generated by this model and the Kalman Filter estimates with the error ellipsis are shown in Figure 7.1. The Kalman filter calculates optimal state estimates with the associated covariance matrices recursively. The key idea is to estimate the latent state as a weighted average of the predicted state and the observed state. We refer reader to [47] for details.

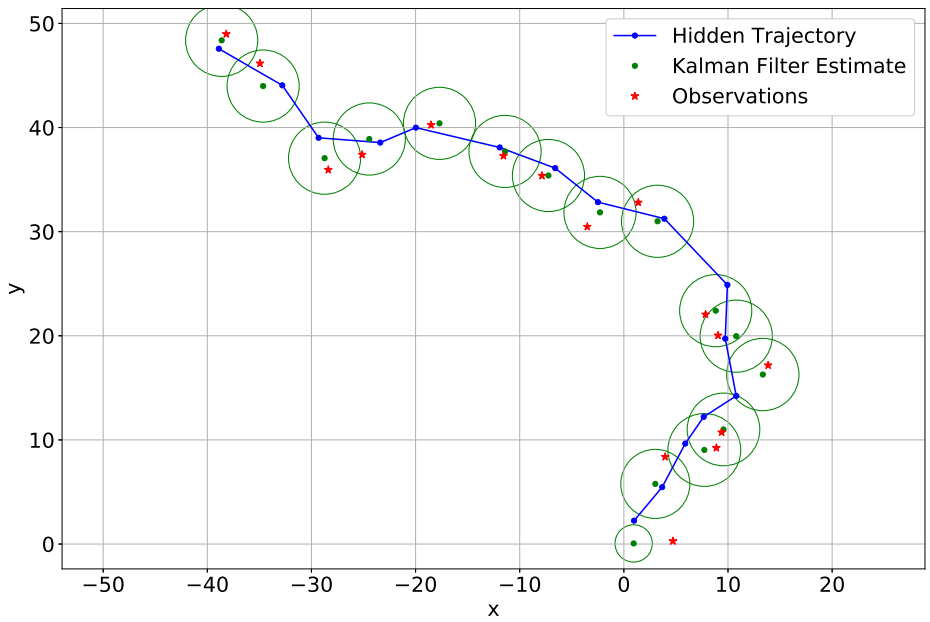

Figure 7.1. An Example of LGSSM Trajectory.

We used the optimal state estimates calculated by the Kalman filter as the ground truth. The part (a) in Figure 7.2 shows a comparison of the mean squared errors (MSE) of particle filtering approximations. The MSE values were computed by averaging over

100 independent simulations, and the number of PEs used is 16.

$$\text{Error} = \frac{1}{T}\sum_{t=1}^{T}\left\|\hat{x}_t^{\text{KF}} - \frac{1}{N}\sum_{i=1}^{N} X_t^i\right\|^2 \tag{7.6}$$

We observe that the errors of the standard multinomial resampling and the RBC algorithm decreases at a similar rate as the number of particles increases. However, the one-pair RBC algorithm converges slower since the interaction between PEs is sparse.

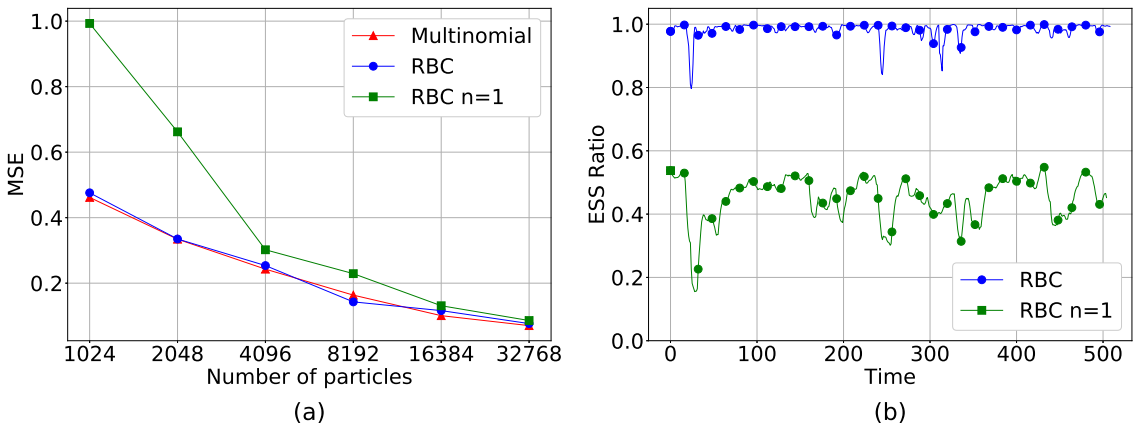

Figure 7.2. MSE and ESS of Resampling Algorithms (LGSSM).

We plot the ESS ratio (ESS/N) of the RBC algorithm for a single simulation in the second part (b) of the Figure 7.2. For better visualization, the ESS ratios are smoothed by a 8-step moving average filter. We ran the simulation on 16 PEs and we used 2^{15} particles. We see that the RBC algorithm keeps the ESS ratio very close to 1. However, when only one pair of PEs are allowed to communicate, the sparse interaction between PEs decreases the ESS and the accuracy of the esimates.

Figure 7.3 shows the execution and resampling time of the algorithms on a cluster system having 16 nodes for 2^{15} particles. We observe the resampling time of the standard approach stops decreasing when the number of PEs (P) is large. The intensive communication between all the PEs clearly hinders the speed-up to be gained through the parallelization. In the RBC algorithm, on the other hand, increasing the number

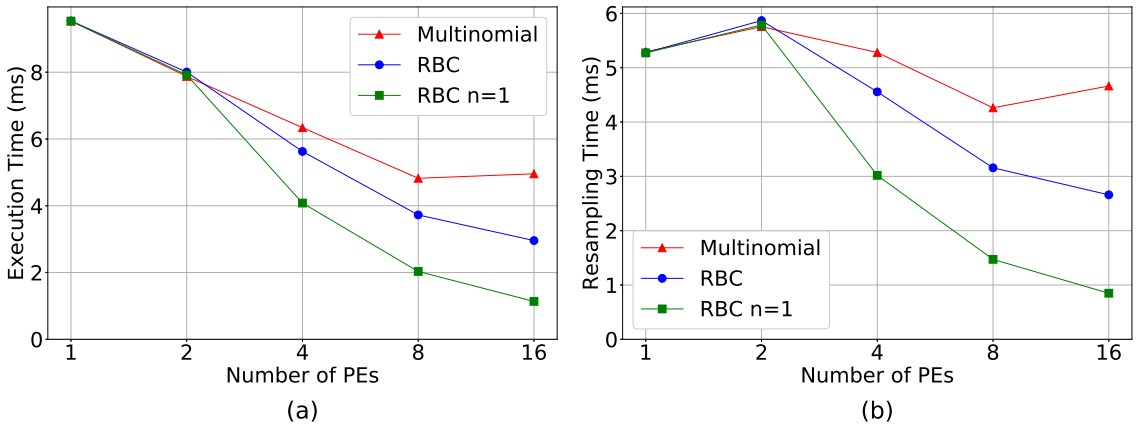

Figure 7.3. Performance of Resampling Algorithms on a Cluster System (LGSSM).

of PEs does not impose an additional communication cost, therefore the resampling time smoothly decreases with the number of PEs. The RBC algorithm with one communicating pair yields the best performance in terms of resampling time, since the communication cost is significantly smaller than the others.

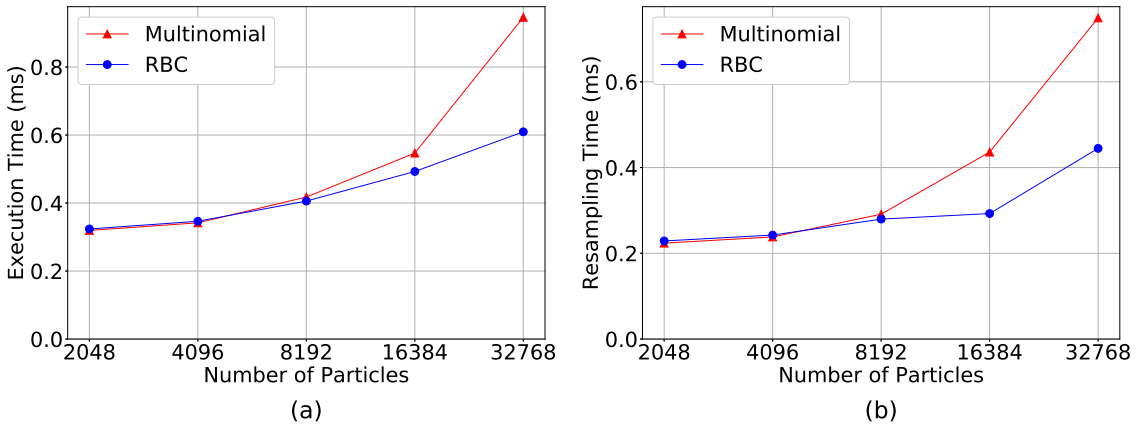

Figure 7.4. Performance Comparison of Resampling Algorithms on a GPU (LGSSM).

In Figure 7.4 we depict the execution and resampling time of the algorithms on a GPU. We observe that the consecutive high latency memory accesses slow down the speed of the standard approach, which eventually limits its scalability. However, the RBC algorithm does not require intensive global memory operations, each thread

block fetches the weights of the particles from the global memory two times during the calculation of the ancestor indices at each time step. Thus, increasing the number of blocks does not have a substantial negative impact on the performance of the RBC algorithm.

7.1.1. Indoor Localization using Bluetooth Low Energy Beacons

SMC filtering can be used to track a mobile sensor receiving Bluetooth Low Energy messages in a building. Bluetooth Low Energy (BLE) technology provides low energy consumption which makes the duration of the transmitters, called beacons, to years. Therefore, BLE beacons are able to be deployed at the positions where the power supply is not availabe. BLE messages also provide Received Signal Strength Indicator (RSSI) to be used in localization.

In an open area, RSSI values received decrease as the distance from the transmitter increases. Unfortunately, this is not the case in buildings. Due to the external factors such as reflection and scattering, RSSI values measured are highly corrupted. One of the prominent methods for indoor localization is fingerprinting which relies on prior scene analysis and works as follows: first, RSSI values are collected for particular locations and corresponding RSSI vectors, also called fingerprints, are constructed, then the location of a sensor is estimated by comparing the RSSI values received by the sensor with the fingerprints. The accuracy of the estimates in this method increases as the grid density of the fingerprints increases. However obtaining dense fingerprints may not be possible due to the installation overhead. There are a couple of methods to overcome this problem such as vector interpolation. A recent approach has been introduced in [46], which increases the number of available fingerprints by exploiting Wasserstein distance interpolation.

We used the data used in [46] in our simulations. The collected fingerprint data consists of RSSI values transmitted by 5 different beacons in a living room for 50 points. This collected data were extended to 50K points using Wasserstein interpolation technique [48]. The distance between two consecutive points is 10 cm. For each grid

point $\mathbf{x} := [x, y]$ where x and y denote the position, we store the RSSI values of each beacon as pairs $\mathbf{z_x}^{(i)} := [r_{\mathbf{x}}^{(i)}, b_{\mathbf{x}}^{(i)}]$. Here $r_{\mathbf{x}}^{(i)}$ and $b_{\mathbf{x}}^{(i)}$ denote the RSSI value and beacon MAC address respectively.

In our observation model, the probability of observing an RSSI of a beacon $\mathbf{z}$ at the position $\mathbf{x}$ is fixed for all t and given by

$$p(\mathbf{Z}_t = \mathbf{z} \mid \mathbf{X}_t = \mathbf{x}) = \frac{1}{|\mathbb{B}|} \frac{\sum_i \delta_{z_{\mathbf{x}}^{(i)} z} \delta_{b_{\mathbf{x}}^{(i)} b}}{\sum_i \delta_{b_{\mathbf{x}}^{(i)} b}}. \tag{7.7}$$

In this equation, $\mathbb{B}$ denotes the set of MAC addressed of all beacons and δ denotes the Kronecker delta. We plot the heatmap of probabilities for a particular beacon and for RSSI values -80 and -85 in Figure 7.5. We can observe that the heatmaps are totally different from each other altough the RSSI values are close.

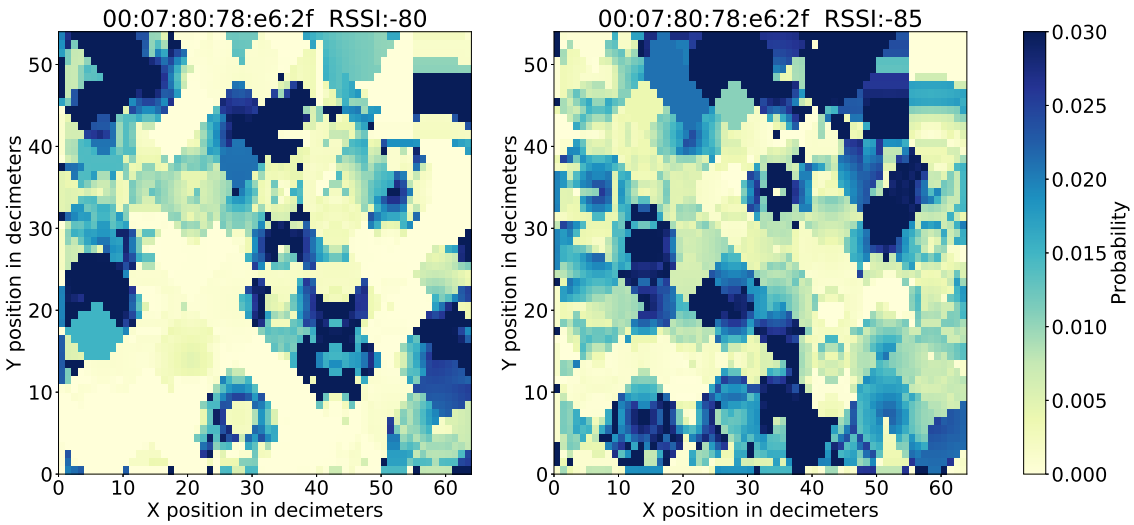

Figure 7.5. An Example of Heatmap of RSSI values.

We calculated the filtering distributions of a moving reciever for each time step using the received RSSI values using the HMM filter [2]. We used a uniform distribution

as the initial distribution, and we adopted the following simple transition probabilities:

$$p(\mathbf{X}_{t+1} = \mathbf{x}_{t+1} \mid \mathbf{X}_t = \mathbf{x}_t) = \frac{\exp\Big(- \frac{||\mathbf{x}_{t+1} - \mathbf{x}_t||^2}{2\sigma^2} \Big)}{\sum_{\mathbf{x} \in \mathbb{X}} \exp\Big(- \frac{||\mathbf{x} - \mathbf{x}_t||^2}{2\sigma^2} \Big)}. \tag{7.8}$$

We display the heatmaps of the calculated filtering distributions with the actual positions for time steps 50 and 100 in Figure 7.6.

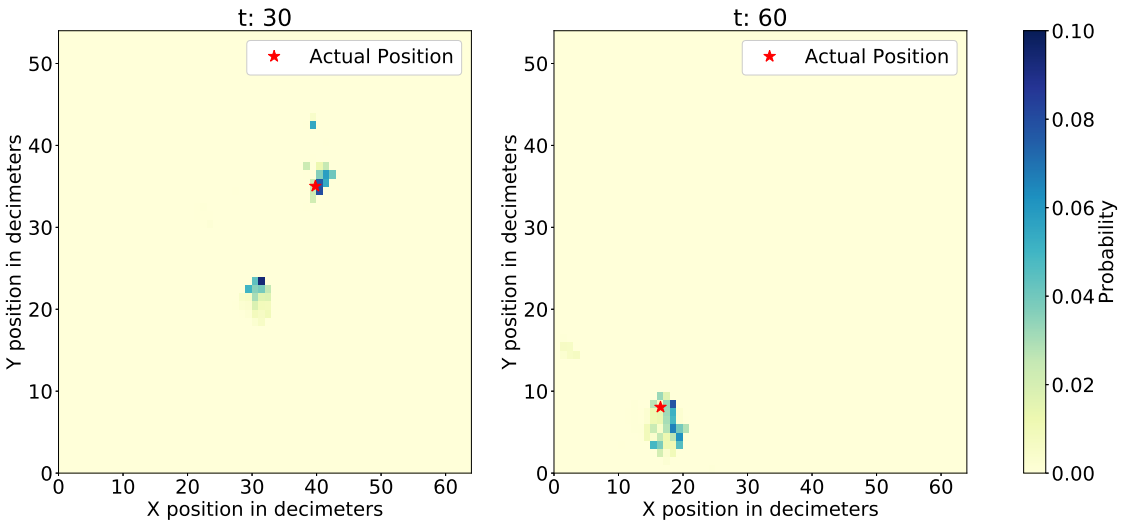

Figure 7.6. Heatmap of Filtering Distributions calculated by the HMM filter.

We used the filtering distributions provided by the HMM filter to evaluate the accuracy of the particle filtering approximations. However, we did not calculate the MSE of the expected positions of the target to avoid potentially misleading results. Instead, we used Wasserstein distance [48] with square loss to evaluate the accuracy of the approximations. We plot the distances in the part (a) of Figure 7.7, and we show the ESS ratio of the algorithms in part (b). We see that there is no significant difference between multinomial resampling and the RBC algorithm.

We display the execution and resampling time of the algorithms on a cluster system in Figure 7.8. The resampling time of the standard multinomial method does not decrease as the number of PEs increases, which eventually degrades the overall

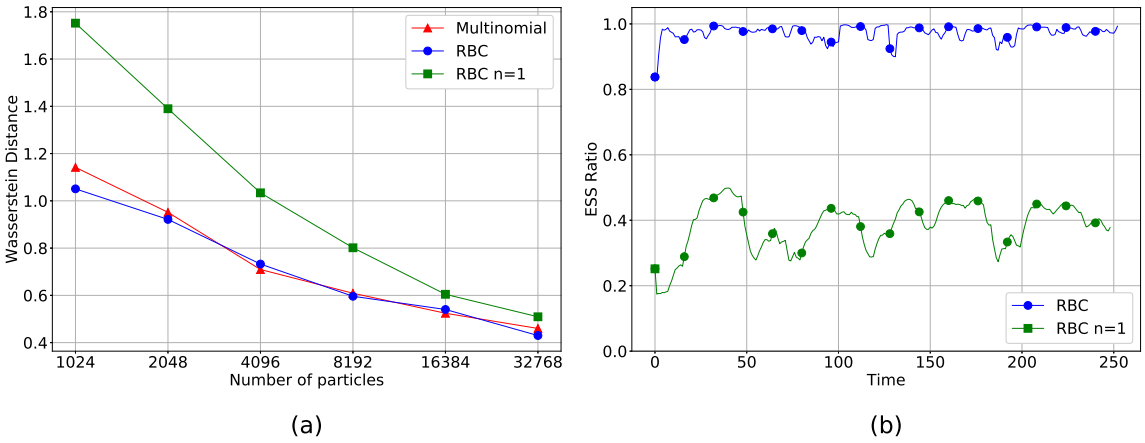

Figure 7.7. Wasserstein Distance and ESS of Resampling Algorithms (Indoor Localization).

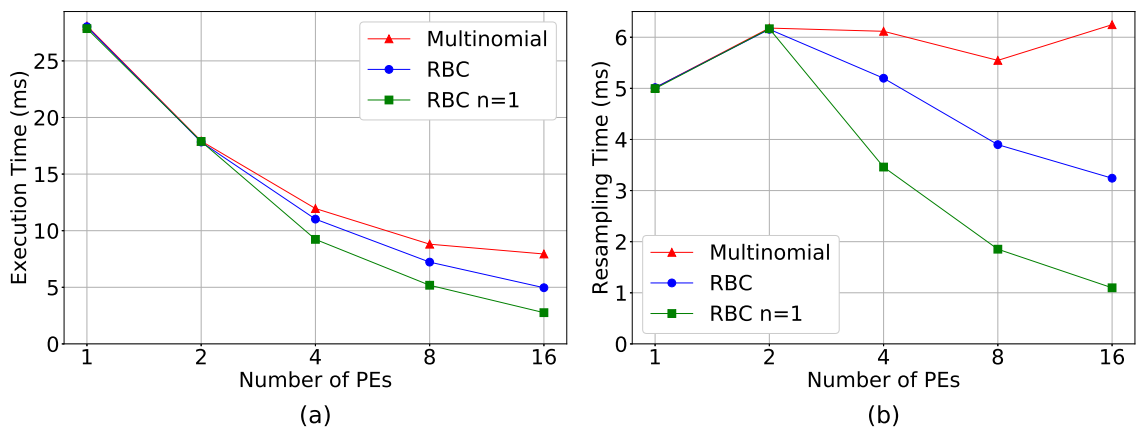

Figure 7.8. Performance Comparison of Resampling Algorithms on a Cluster System (Indoor Localization).

performance, whereas the resampling time of the RBC algorithms decreases almost linearly. Figure 7.9 shows the comparison of the performance of the algorithms on a GPU. We can see that though the propagation and update operations take most of the time in this application, the RBC algorithm still provides significant performance gain.

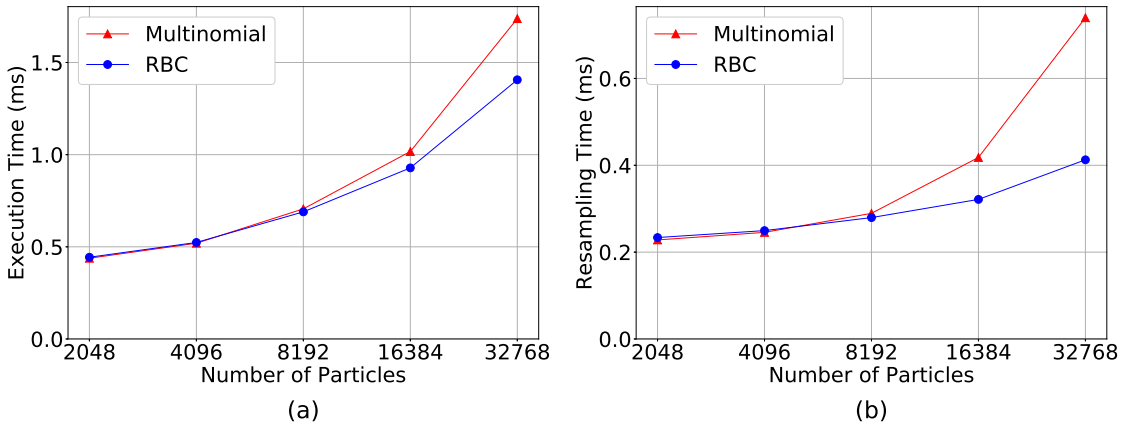

Figure 7.9. Performance Comparison of Resampling Algorithms on a GPU (Indoor Localization).

7.1.2. Lorenz System

We used the application of tracking the state of the chaotic Lorenz system to evaluate our methods. The Lorenz system is non-periodic and highly nonlinear, and small errors in its estimated state result in larger errors in a later state [49]. We used the discrete time version as the state transition model in our simulations described in [50]. The state vector is 3-dimensional $(x_{1,t}, x_{2,t}, x_{3,t})$ and it is updated over time according to the following propagation equations:

$$x_{1,t} = x_{1,t-1} - ST(x_{1,t-1} - x_{2,t-1}) + \sqrt{T}u_{1,t} \tag{7.9}$$

$$x_{2,t} = x_{2,t-1} + T(Rx_{1,t-1} - x_{2,t-1} - x_{1,t-1}x_{3,t-1}) + \sqrt{T}u_{2,t} \tag{7.10}$$

$$x_{3,t} = x_{3,t-1} - T(Bx_{3,t-1} - x_{1,t-1}x_{2,t-1}) + \sqrt{T}u_{3,t}. \tag{7.11}$$

We can only make noisy measurements on the first dimension every 4 time steps.

$$z_t = x_{1,t} + \nu_k \tag{7.12}$$

S, R, B are the system constants and their default values are $10, 28, 8/3$ respectively. T is the time step which equals 0.025 time units and $\nu_t, u_{i,t}$ are zero-mean Gaussian white noise with the variance $\sigma_\nu^2 = 0.01$ and $\sigma_u^2 = 1$ respectively.. The initial state is distributed from a Gaussian distribution with the standard mean (-5.91652, -5.52332, 24.5723) and the covariance $10^{-2} \cdot I_3$. Figure 7.10 shows the generated trajectory and the measurements on the first dimension.

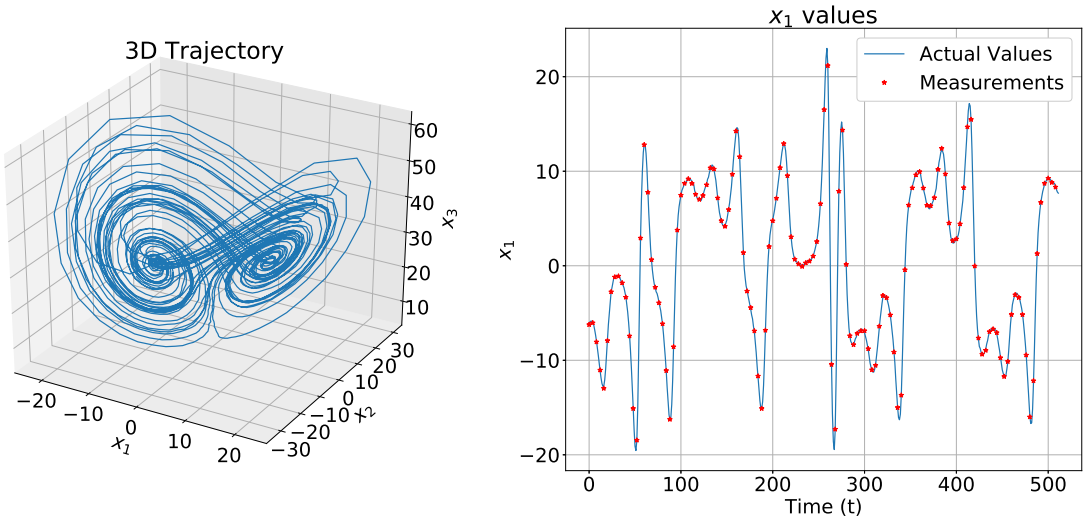

Figure 7.10. An Example of Lorenz Trajectory.

We cannot calculate the optimal filtering distributions for the Lorenz system since it does not have a finite state space, and it is nonlinear and non-Gaussian. Therefore we calculated the MSE values of the state approximations using directly the actual states.

$$\text{Error} = \frac{1}{T}\sum_{t=1}^{T}\left\| x_t - \frac{1}{N}\sum_{i=1}^{N} X_t^i \right\|^2 \tag{7.13}$$

The MSE values of the algorithms are shown in the part (a) of Figure 7.11, and the ESS ratios of the RBC algorithms are shown in part (b). We see that MSE curves of the standard multinomial resampling and the RBC algorithm are so close that they overlap, which confirms that the complete interaction is indeed not needed for many models including highly nonlinear models such as the Lorenz System.

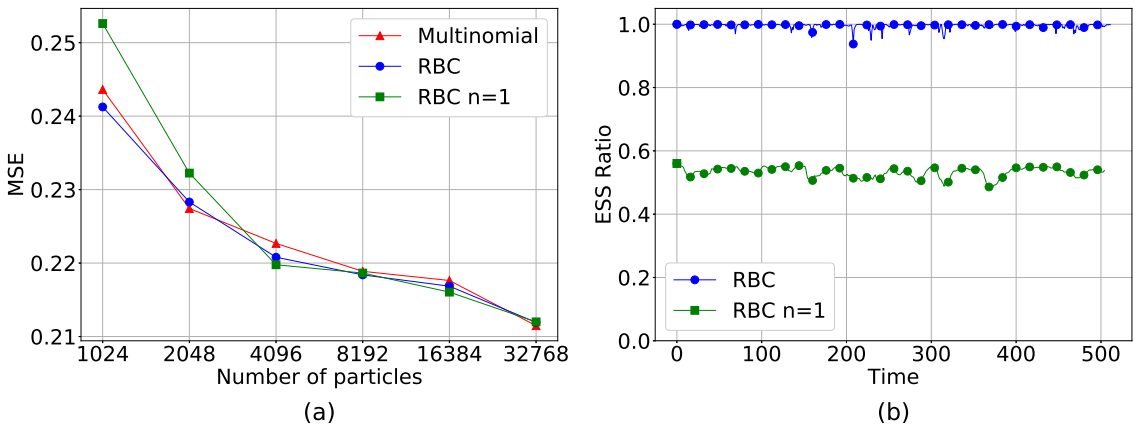

Figure 7.11. MSE and ESS of Resampling Algorithms (Lorenz).

The performance comparison of the algorithms on a cluster system and on a GPU are shown in Figure 7.12 and Figure 7.13 respectively. The results are similar to those of the LGSSM and the indoor localization applications. We can observe in Figure 7.12 that the RBC algorithm outperforms the standard approach, and the RBC with one communicating pair has the least resampling time. Similarly, we can also see in Figure 7.13 that the global memory access term in Equation 5.5 becomes dominant as the number of particles (therefore blocks) increases, hence the standard multinomial resampling performs poorly compared to the RBC algorithm for large number of particles.

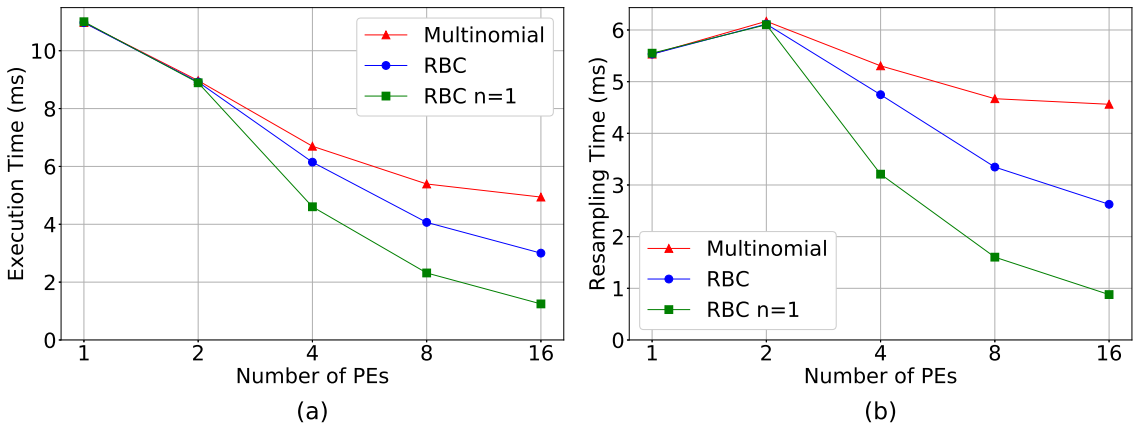

Figure 7.12. Performance Comparison of Resampling Algorithms on a Cluster System (Lorenz).

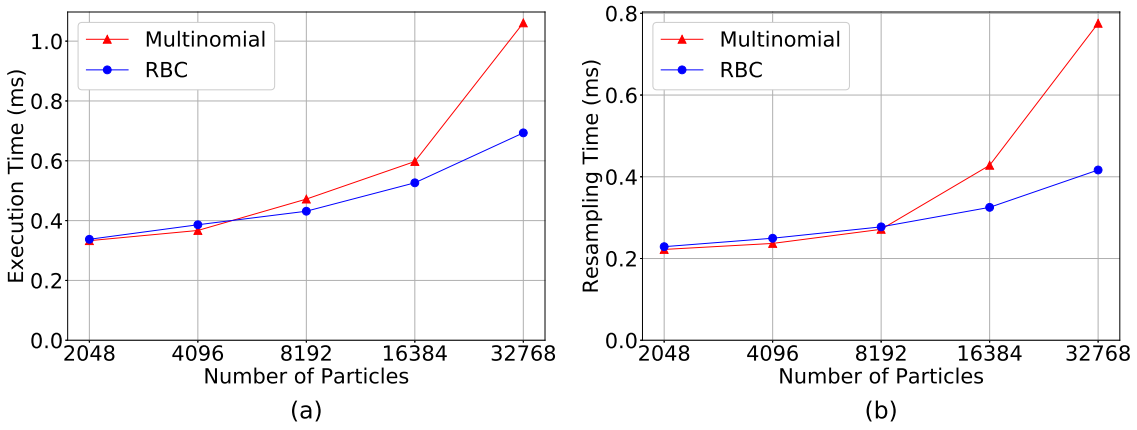

Figure 7.13. Performance Comparison of Resampling Algorithms on a GPU (Lorenz).

8. CONCLUSION

In this thesis, we investigated the implementation of parallel resampling algorithms on different computing architectures. In Chapter 3 we discussed the key aspects of parallel computing architectures including computer cluster systems, shared memory architectures and GPUs, which can affect the speed of a parallel resampling algorithm. Based on these considerations, we presented a standard way to parallelize resampling algorithms in Chapter 5. We then described the bottlenecks of the standard approach, and we provided mathematical formulations for the execution time on both cluster systems and GPUs.

We showed that the standard parallel resampling algorithm suffers from two main problems in a cluster system. First, the consecutive message transmissions and receptions incur a significant overhead as the number of processing elements increases. Second, the standard method performs poorly when the interconnection network has high latency. We also demonstrated intensive global memory operations performed by the standard method limits its scalability on a GPU.

In Chapter 6 we proposed, a novel parallel resampling algorithm, resampling with butterfly communications (RBC), inspired by butterfly resampling previously described in [14]. The RBC algorithm puts some constraints on the communication pattern according to the butterfly structure such that the nodes communicate only in pairs. We showed that the RBC algorithm is an instance of augmented resampling, and it satisfies the stability conditions. We also showed the RBC algorithm eliminates the important limitations of the standard approach.

In the RBC algorithm, the number of received or transmitted messages does not increase with number of processing elements, because each processing element communicates only one other processing element during the entire resampling operation. Similarly, the thread blocks access at most two separate global memory locations during kernel execution on a GPU.

We further proposed a parameterized version of the RBC algorithm by the number of communicating pairs during resampling in order to alleviate the high network latency. As the number of communicating pairs gets smaller, the effect of the high latency is reduced. However, the sparse interaction between the processing elements makes the RBC algorithm less stable and may result in loss of accuracy. The appropriate choice of the number of communicating pairs to balance the tradeoff between accuracy and speed depends on the transition and the observation model, as well as the discrepancy between the target and the proposal distribution.

We reported the experimental results in Chapter 7. We ran simulations on a GPU with CUDA compute capability 6.1 and a computer cluster system having 16 server nodes for three particle filtering applications: tracking of the position of a target moving within x-y plane according to a simple LGSSM, indoor localization using BLE Beacons, and tracking the state of the chaotic Lorenz system. We found that the standard approach fails to scale due to the intensive communications it requires. On the other hand, the RBC algorithm provides substantial speed-up while keeping the accuracy of the estimates significantly high. In addition, decreasing the number of communicating pairs further increases the speed-up in exchange for negligible loss of accuracy.

REFERENCES

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

2. Rabiner, L. R., "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition", *Proceedings of the IEEE*, Vol. 77, No. 2, pp. 257–286, 1989.

3. Doucet, A., N. de Freitas and N. Gordon, "An Introduction to Sequential Monte Carlo Methods", A. Doucet, N. de Freitas and N. Gordon (Editors), *Sequential Monte Carlo Methods in Practice*, pp. 3–14, Springer New York, New York, NY, USA, 2001.

4. Julier, S. J. and J. K. Uhlmann, "A New Extension of the Kalman Filter to Nonlinear Systems", *The 11th International Symposium on Aerospace/Defense Sensing, Simulation and Controls*, Vol. 3068, pp. 182–193, 1997.

5. Doucet, A. and A. M. Johansen, "A tutorial on particle filtering and smoothing: Fifteen years later", D. Crisan and B. Rozovskii (Editors), *Handbook of Nonlinear Filtering*, pp. 656–704, Oxford University Press, New York, NY, USA, 2011.

6. Crisan, D., "Particle Filters - A Theoretical Perspective", A. Doucet, N. de Freitas and N. Gordon (Editors), *Sequential Monte Carlo Methods in Practice*, pp. 3–14, Springer New York, New York, NY, USA, 2001.

7. Lin, M., R. Chen and J. S. Liu, "Lookahead Strategies for Sequential Monte Carlo", *Statistical Science*, Vol. 28, No. 1, pp. 69–94, 2013.

8. Schon, T., F. Gustafsson and P. J. Nordlund, "Marginalized Particle Filters for Mixed Linear/Nonlinear State-Space Models", *IEEE Transactions on Signal Processing*, Vol. 53, No. 7, pp. 2279–2289, 2005.

9. Briers, M., A. Doucet and S. Maskell, "Smoothing Algorithms for State-Space Models", *Annals of the Institute of Statistical Mathematics*, Vol. 62, No. 1, p. 61, 2009.

10. Andrieu, C., A. Doucet and R. Holenstein, "Particle Markov chain Monte Carlo methods", *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, Vol. 72, No. 3, pp. 269–342, 2010.

11. Naesseth, C. A., S. W. Linderman, R. Ranganath and D. M. Blei, "Variational Sequential Monte Carlo", *ArXiv e-prints*, 2017, `https://arxiv.org/pdf/1705.11140.pdf`, accessed at May 2018.

12. Sidenbladh, H., "Multi-Target Particle Filtering for the Probability Hypothesis Density", *Proceedings of the Sixth International Conference of Information Fusion*, Vol. 2, pp. 800–806, 2003.

13. Vo, B.-N., S. Singh and A. Doucet, "Sequential Monte Carlo Implementation of the PHD Filter for Multi-target Tracking", *Proceedings of the Sixth International Conference of Information Fusion*, Vol. 2, pp. 792–799, 2003.

14. Heine, K., N. Whiteley, A. T. Cemgil and H. Guldas, "Butterfly Resampling: Asymptotics for Particle Filters with Constrained Interactions", *ArXiv e-prints*, 2014, `http://arxiv.org/abs/1411.5876`, accessed at May 2018.

15. Guldas, H., A. T. Cemgil, N. Whiteley and K. Heine, "A practical introduction to butterfly and adaptive resampling in Sequential Monte Carlo", *IFAC-PapersOnLine*, Vol. 48, No. 28, pp. 787–792, 2015.

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

17. Bashi, A. S., V. P. Jilkov, X. R. Li and H. Chen, "Distributed Implementations of Particle Filters", *Proceedings of the Sixth International Conference of Information*

Fusion, Vol. 2, pp. 1164–1171, 2003.

18. Bolic, M., P. M. Djuric and S. Hong, "Resampling Algorithms and Architectures for Distributed Particle Filters", *IEEE Transactions on Signal Processing*, Vol. 53, No. 7, pp. 2442–2450, 2005.

19. Sutharsan, S., T. Kirubarajan, T. Lang and M. Mcdonald, "An Optimization-Based Parallel Particle Filter for Multitarget Tracking", *IEEE Transactions on Aerospace and Electronic Systems*, Vol. 48, No. 2, pp. 1601–1618, 2012.

20. Balasingam, B., M. Bolić, P. M. Djurić and J. Míguez, "Efficient Distributed Resampling for Particle Filters", *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 3772–3775, 2011.

21. Chitchian, M., A. Simonetto, A. S. van Amesfoort and T. Keviczky, "Distributed Computation Particle Filters on GPU Architectures for Real-Time Control Applications", *IEEE Transactions on Control Systems Technology*, Vol. 21, No. 6, pp. 2224–2238, 2013.

22. Demirel, Ö., I. Smal, W. Niessen, E. Meijering and I. F. Sbalzarini, "PPF - A Parallel Particle Filtering Library", *The 10th Data Fusion & Target Tracking Conference*, pp. 52–59, 2014.

23. MPI Forum, *A Message-Passing Interface Standard Version 3.1*, `https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`, accessed at May 2018.

24. Miao, L., J. J. Zhang, C. Chakrabarti and A. Papandreou-Suppappola, "A New Parallel Implementation for Particle Filters and Its Application to Adaptive Waveform Design", *2010 IEEE Workshop On Signal Processing Systems*, pp. 19–24, 2010.

25. 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*, Vol. 25, No. 2, pp. 243–260, 2015.

26. Whiteley, N., A. Lee and K. Heine, "On the role of interaction in sequential Monte Carlo algorithms", *Bernoulli*, Vol. 22, No. 1, pp. 494–529, 2016.

27. Lee, A. and N. Whiteley, "Forest resampling for distributed sequential Monte Carlo", *Statistical Analysis and Data Mining: The ASA Data Science Journal*, Vol. 9, No. 4, pp. 230–248, 2016.

28. Maskell, S., B. Alun-Jones and M. Macleod, "A Single Instruction Multiple Data Particle Filter", *2006 IEEE Nonlinear Statistical Signal Processing Workshop*, pp. 51–54, 2006.

29. Hendeby, G., J. D. Hol, R. Karlsson and F. Gustafsson, "A Graphics Processing Unit Implementation of the Particle Filter", *2007 15th European Signal Processing Conference*, pp. 1639–1643, 2007.

30. Blelloch, G. E., *Prefix Sums and Their Applications*, Tech. Rep. CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 1990.

31. Nvidia Corporation, *CUDA C Programming Guide*, 2018, `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`, accessed at May 2018.

32. Chao, M. A., C. Y. Chu, C. H. Chao and A. Y. Wu, "Efficient parallelized particle filter design on CUDA", *2010 IEEE Workshop On Signal Processing Systems*, pp. 299–304, 2010.

33. Gong, P., Y. O. Basciftci and F. Ozguner, "A Parallel Resampling Algorithm for Particle Filtering on Shared-Memory Architectures", *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, pp. 1477–

1483, 2012.

34. Murray, L., "GPU acceleration of the particle filter: the Metropolis resampler", *ArXiv e-prints*, 2012, `https://arxiv.org/pdf/1202.6163.pdf`, accessed at May 2018.

35. Murray, L. M., A. Lee and P. E. Jacob, "Parallel Resampling in the Particle Filter", *Journal of Computational and Graphical Statistics*, Vol. 25, No. 3, pp. 789–805, 2016.

36. Rauber, T. and G. Rünger, *Parallel Programming: for Multicore and Cluster Systems*, Springer-Verlag Berlin Heidelberg, Berlin, Germany, 2013.

37. Amazon.com Inc., *An Introduction to High Performance Computing on AWS*, 2015, `https://d0.awsstatic.com/whitepapers/Intro_to_HPC_on_AWS.pdf`, accessed at May 2018.

38. Microsoft Corporation, *High Performance Computing on Microsoft Azure for Scientific and Technical Applications*, 2016, `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/04/high-perf-computing-on-microsoft-azure.pdf`, accessed at May 2018.

39. Culler, D., R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian and T. von Eicken, "LogP: Towards a Realistic Model of Parallel Computation", *SIGPLAN Not.*, Vol. 28, No. 7, pp. 1–12, 1993.

40. Karp, R. M., A. Sahay, E. E. Santos and K. E. Schauser, "Optimal Broadcast and Summation in the LogP Model", *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 142–153, 1993.

41. Kirk, D. B. and W.-M. W. Hwu, *Programming Massively Parallel Processors (Third Edition)*, Morgan Kaufmann, Cambridge, MA, USA, 2017.

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

43. Gordon, N. J., D. J. Salmond and A. F. M. Smith, "Novel approach to nonlinear/non-Gaussian Bayesian state estimation", *IEE Proceedings F - Radar and Signal Processing*, Vol. 140, No. 2, pp. 107–113, 1993.

44. *Numba*, `https://numba.pydata.org/`, accessed at May 2018.

45. *MPI for Python*, `https://bitbucket.org/mpi4py/mpi4py`, accessed at May 2018.

46. Daniş, F. S. and A. T. Cemgil, "Model-Based Localization and Tracking Using Bluetooth Low-Energy Beacons", *Sensors*, Vol. 17, No. 11, p. 2484, 2017.

47. Särkkä, S., *Bayesian Filtering and Smoothing*, Cambridge University Press, Cambridge CB2 8BS, UK, 2013.

48. Villani, C., *Optimal Transport: Old and New*, Springer-Verlag Berlin Heidelberg, Berlin, Germany, 2008.

49. Lorenz, E. N., "Deterministic Nonperiodic Flow", *Journal of the Atmospheric Sciences*, Vol. 20, No. 2, pp. 130–141, 1963.

50. Míguez, J., "Analysis of Parallelizable Resampling Algorithms for Particle Filtering", *Signal Processing*, Vol. 87, No. 12, pp. 3155–3174, 2007.