Parallelised Bayesian Optimisation for Deep Learning

Thesis submitted in accordance with the requirements of
the University of Liverpool for the degree of Doctor of Philosophy by

Lykourgos Kekempanos

January 29, 2019

Contents

Illustrations

List of Figures

List of Tables

Abbreviations

The following abbreviations are found throughout this thesis:

API	Application Programming Interface
APL	Array Programming Language
ATZ	All-Trailing-Zeros
EM	Expectation-Maximization
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
HDFS	Hadoop Distributed File System
HMC	Hamiltonian (or Hybrid) Monte Carlo
HPC	High Performance Computing
MALA	Metropolis-Adjusted Langevin Algorithm
MCMC	Markov Chain Monte Carlo
MH	Metropolis-Hastings
MLE	Maximum Likelihood Estimation
MLP	Multi-Layer Perceptrons
MPI	Message Passing Interface
MSE	Mean Squared Error
PDF	Probability Density Function
PE	Processing Element
PILL	Partially Implicit Local Linearisation
PPS	Particles Processed per Second
RBF	Radial Basis Function
RBM	Restricted Boltzmann Machine
RDD	Resilient Distributed Dataset
RJMCMC	Reversible Jump Markov Chain Monte Carlo
RMSE	Root Mean Square Error
RNA	Resampling with Nonproportional Allocation
RPA	Resampling with Proportional Allocation
RWMH	Random Walk Metropolis-Hastings

SIR	Sequential Importance Resampling
SIS	Sequential Importance Sampling
SLP	Single Layer Perceptron
SMC	Sequential Monte Carlo
TMCMC	Transitional Markov Chain Monte Carlo
VLSI	Very-Large-Scale Integration

Abstract

Training of deep neural networks (DNN) is an indispensable process in machine learning. The training process of DNNs aims to optimise the parameter values of the network, often relies on the derivative of the log-likelihoods of the underlying parameter space. As such, it is highly probable that the optimisation process to find local optimum values instead of the global ones. In addition to this, conventional approaches used for this process, such as Markov chain Monte Carlo methods, not only offer suboptimal runtime performance, but also prevent effective parallelisation due to inherent dependencies in the process.

In this thesis, we consider an alternative approach to Markov chain Monte Carlo (MCMC) methods, namely the Sequential Monte Carlo (SMC) sampler, which generalises particle filters. More specifically, the thesis focuses on improving the performance and accuracy of the SMC methods, particularly in the context of fully Bayesian learning.

The Radial Basis Function (RBF) network is an example of such training process based on fully Bayesian learning. In this setting, the thesis proposes a new method to train neural networks using the importance sampling and resampling. The initial comparison of the two methods reveal that the proposed methodology is worse in both terms of accuracy and performance. This lead the research to concentrate of the performance and accuracy improvements of the proposed approach.

The performance analysis began with application of a new proposed, parallel and fully distributed resampling methodology, with improved time complexity than the original approach using two MapReduce frameworks, Hadoop and Spark. Results indicate that Spark is up to 25 times faster than Hadoop, while on Spark the new proposed methodology is up to 10 times faster than the original method. However, it is noticed that application of the same algorithm on Message Passing Interface (MPI) provide significantly better runtimes and is more suitable for the proposed algorithm.

The accuracy analysis began with experiments illustrating that the basic Sequential Monte Carlo sampler provides worse accuracy than alternative or competitor MCMC algorithms. Three different strategies are applied on the basic Sequential Monte Carlo sampler providing better accuracy. The analysis is extended to include competitor algorithms. The exhaustive evaluation shows that the proposed approach offers superior performance and accuracy.

Acknowledgements

I am profoundly grateful to my supervisors Prof. Simon Maskell, Dr Jeyarajan Thiyagalingam, and Dr Yannis Goulermas for their support, guidance, and advice throughout my PhD. During this collaboration, I appreciated their scientific approach to research problems.

I thank my advisors Prof. Jason Ralph, Dr Tingting Mu, and Prof. Prudence Wong for their suggestions and insights during my PhD. I appreciate the discussions and the comments I received from Dr Peter Green. I would like to thank my viva examiners, Dr. Francisco Alejandro Díaz De la O and Dr. Gustaf Hendeby for their suggestions that were very helpful in submitting this thesis.

I gratefully acknowledge the UK EPSRC Doctoral Training Award, the support of STFC Daresbury, and STFC Hartree Centre for providing us with the computational resources for this work.

I thank Alessandro Varsi for the discussions and collaboration. Special mention to other members of the research group including, Alex Phillips, Chinmay Mishra, Chloe Barrett-Pink, Chongyang Liu, Elpida Kontsioti, Flávio De Melo, James Wright, Joanna Hajne, Josh Coates, Lee Devlin, Luydmil Vladimirov, Matteo Fasiolo, Paul Horridge, Richard Sloane, Robert Moore, Roberta Piroddi, and Yifan Zhou.

This research contributed to the EPSRC 2.5 million (pounds) grant with the title "Big Hypotheses: A Fully Parallelised Bayesian Inference Solution" with reference number "EP/R018537/1" and principal investigator, Simon Maskell, who was my primary supervisor.

Η διδακτορική μου διατριβή είναι αφιερωμένη στους γονείς μου, Κωνσταντίνο και Αναστασία, και τις αδερφές μου Σοφία, Κυριακή, Αθανασία και Παναγιώτα για την άνευ όρων αγάπη και υποστήριξη. Σας ευχαριστώ πολύ.

Chapter 1

Introduction

1.1 Overview

Deep learning and neural networks are machine learning algorithms designed to make predictions based on extracted features or filters from a dataset. Deep learning, or deep neural network, is an extension of neural networks by containing more than one hidden layer. Medical image analysis [57], object classification [74], and natural language processing [107] are a small sample of many application domains. Deep learning and neural networks are categorised as deterministic or stochastic according to the applied training procedure. A detailed discussion on the training process of deterministic and stochastic deep learning methods is provided in Chapter 3.

Maximum Likelihood Estimation (MLE) (e.g., [26]) and Maximum a Posteriori (MAP) (e.g. [104]) are two widely applicable approaches to train the network. MLE first derives the log-likelihood of the model and then maximises it with regard to the parameters of the model with an optimisation algorithm. Typically, stochastic gradient ascent is the optimisation method applied. An example of MLE is the restricted Boltzmann machine using the contrastive divergence [8]. The computation of the derivative of the log-likelihood requires the computation of expectations, which cannot be calculated analytically but are approximated using sampling or Markov chain Monte Carlo (MCMC) methods. The stochastic gradient descent with traditional MCMC methods, such as the Metropolis-Hastings (MH) or the Gibbs sampling algorithms, can lead to poor local minimum solutions and is challenging to tune [26]. Apart from the accuracy aspects, it is also hard to parallelise traditional MCMC methods. For instance, MH is a sequential algorithm constructing a Markov chain where the current sample is depended on the exact preceding sample.

An alternative methodology is full Bayesian learning, which requires the computation of the full posterior distribution over all possible parameter settings of which the Radial Basis Function (RBF) network in an example [10]. This thesis aims to apply Bayesian inference using a Sequential Monte Carlo (SMC) method (e.g., the SMC sampler) as an alternative algorithm to the traditional MCMC methodologies, and reveal its benefits and potential to be applied in the training process of neural networks and deep learning algorithms.

Bayesian inference is a sophisticated statistical inference method processed through a combination of the user-defined prior density and uncertain evidence. Bayesian statistics provides a complete picture of the uncertainty in the estimation of the unknown parameters of a model [78] and is not prone to overfitting depending on the user-defined prior density [109].

The Sequential Monte Carlo (SMC) sampler belongs to a wider class of SMC methods. The more widely known Particle filters also belong to this class of methodologies and are applicable in dynamic statistical inference, which refers to drawing conclusions or estimations on time-dependent models (or time evolving models). Such models can have non-linear and non-Gaussian characteristics. In the literature, another set of widely known alternative methods include the Kalman filters and its extended versions. Kalman filters are optimal to linear Gaussian systems.

SMC methods are applied in many domains and real-world problems, such as in robotics to solve problems related with localisation and mapping [99] as well as in finance for stochastic volatility models and estimating dynamic microeconomic models [25], [64]. Other domains are medicine [85], wildfire spread simulation [14].

The thesis begins with background knowledge for understanding the training processes with widely applicable methods, such as the stacked autoencoder and the deep belief network, a deterministic and stochastic deep learning algorithm, respectively. The training procedure in the RBF network uses the MH algorithm as a core method to update the centres of the RBF, while the parameters and hyperparameters of the network are updated with Gibbs sampling. The overall method in [10] is referred to as Hybrid MCMC. A new method is proposed where the MH is replaced with the core methods of importance sampling and resampling used by any SMC method to improve the accuracy and performance. Since the MH algorithm is a sequential algorithm, the replacement with an SMC method offers great potential for enhancing overall performance due to its appealing property that, as the number of samples increases, the ability of the samples to represent the probability density function (pdf) increases and the accuracy of estimates derived from the particles improve. Interestingly, a benchmark comparison of the basic SMC sampler with competitor methodologies does not perform as well as expected, which guided this research to concentrate on delivering a more efficient SMC sampler in terms of performance and accuracy.

Establishing a better training process for neural networks and deep learning algorithms requires multiple preliminary and mandatory steps to improve the overall efficiency of the basic SMC sampler. A key step is a new fully distributed resampling to accelerate the overall performance of any SMC method. Another step is the application of new strategies to improve the accuracy of the method, which help to outperform the basic SMC sampler and competitor methodologies.

1.2 Contributions

The first novel contribution of the thesis is the replacement of the MH with steps of importance sampling and resampling in the training process of the RBF network. Second, is the application of a fully distributed resampling algorithm with better time complexity than available from previous related research. Accuracy improvements of the SMC sampler include the application of a new recycling method compared with the basic approach and the original methodology. Third, the thesis describes the application of alternative forward Markov kernels and a more efficient backward Markov kernel to the traditional random walk. Finally, the proposed SMC sampler is compared with other MCMC methods. These contributions are summarised in more detail in the following:

- A detailed description of all simulations and implementations.
- Contribution to the writing of the published and submitted articles listed below.
- In Section 3.3 the replacement of the Metropolis-Hastings algorithm to steps of importance sampling and resampling.
- In Chapter 4 the review on the parallel resampling.
- In Chapter 5 the application of the Ackley function as a potential new benchmark distribution for the evaluation of the Sequential Monte Carlo (SMC) sampler.
- In Chapter 6 the application of the partially implicit local linearisation method as an alternative proposal distribution to the random walk.
- In Chapters 5, 6, 7 the evaluation procedure of the proposed strategies.

Contributions made by my supervisors or other members of the group; Contribution to the writing of the published and submitted articles listed below by Simon Maskell and Jeyarajan Thiyagalingam; Contribution to the writing of the published articles listed below by Alessanrdo Varsi; In Chapter 4 the new redistribute algorithm with time complexity $O((logN)^2)$ by Simon Maskell; In Chapter 4 the strategy and the evaluation procedure for the new redistribute method by Jeyarajan Thiyagalingam; In Chapter 5 the new recycling method by Simon Maskell; In Chapter 6 the Euler discretisation by Simon Maskell; In Chapter 7 the optimal backward kernel by Simon Maskell; Implementation of the method in the second published article listed below (#2) or the Appendix A by Alessanrdo Varsi. These contributions are based on the following published and submitted articles:

1. "MapReduce Particle Filtering with Exact Resampling and Deterministic Runtime", J. Thiyagalingam, L. Kekempanos, S. Maskell. S, EURASIP Journal on Advances in Signal Processing, 2017.
   I contributed to the writing of the article and implementation of all the algorithms on Hadoop and Spark.

2. "Parallelising Particle Filtering for Deterministic Runtimes on Distributed Memory Systems" A. Varsi, L. Kekempanos, J. Thiyagalingam, S. Maskell, 3rd International Conference on Intelligent Signal Processing, 2017.
   I contributed to the writing of the article, explaining and elaborating all the ideas from MapReduce to MPI.

3. "SMC Samplers and Particle Filters on MPI: an optimised parallel algorithm with $O\left((\log_2 N)^2\right)$ time complexity", A. Varsi, L. Kekempanos, J. Thiyagalingam, S. Maskell. In preparation for submission to the IEEE Transactions on Signal Processing.
   I contributed to the writing of the article and the sequential - benchmark - experiments.

4. "Using an SMC Sampler with a Langevin Proposal as an Efficient Alternative to MALA", S. Maskell, L. Kekempanos, P. Green, M. Fasiolo, F. Melo, J. Thiyagalingam. In preparation for submission to the IEEE Transactions on Signal Processing.
   I contributed to the writing of the article and to the evaluation procedure of the proposed strategies. I proposed the Ackley function as a new benchmark for the experiments and the partially implicit local linearisation as proposal to the SMC sampler. I proposed new - more efficient - strategies to make estimations on bimodal distributions.

1.3 Outline

The thesis follows in six chapters with Chapter 2 including an introduction to Bayesian inference. Several traditional Markov chain Monte Carlo methods and Sequential Monte Carlo methods are discussed with comparisons. Chapter 3 discusses two existing deep learning algorithms, a deterministic and a stochastic network, applied to a human age prediction problem. A new methodology for training a neural network is proposed based on importance sampling and resampling. Chapter 4 provides details on a new proposed algorithm for parallelising the resampling algorithm with the goal to convert the resampling into a more amenable algorithm for a distributed implementation. The proposed method is applied in MapReduce frameworks, while a later version of the method is applied to High Performance Computing (HPC). Chapters 5, 6, and 7 focus on strategies to improve the accuracy of the SMC sampler.

Each chapter corresponds to a proposed strategy. In Chapter 5, all the particles of the SMC sampler are combined to have estimates over multiple iterations. In Chapter 6, Langevin based proposal distributions are applied with the SMC sampler to improve the quality of the generated samples over the random walk proposal assumed in the basic SMC sampler. In Chapter 7, the optimal backward Markov kernel is proposed with respect to the selected forward Markov kernel. Simulations demonstrate the benefits of each strategy compared to the basic SMC sampler, which is followed by its comparison with competitor MCMC methods.

Chapter 2

Bayesian inference and Markov chain Monte Carlo methods

2.1 Introduction

Markov chain Monte Carlo (MCMC) methods are used to estimate the posterior density function (analytical computation is intractable [100]) as computed using Bayesian inference. This chapter provides a brief introduction on such methods beginning with a formal explanation of Bayesian inference followed by some MCMC methods, including the Metropolis-Hastings (MH), Hamiltonian Monte Carlo (HMC), Transitional Markov Chain Monte Carlo (TMCMC), and Sequential Monte Carlo (SMC) methods. This overview offers a reference for the following chapters as the methods discussed are applied in simulations and experiments.

2.2 Bayesian inference

Assuming a model with a vector of unknown parameters (hypothesis) x after observations (data or evidence) D, Bayesian inference correlates the posterior distribution, denoted $p(x|D)$, with the prior probability density function $p(x)$. The conditional probability of D given x, $p(D|x)$ is known as the likelihood probability density function and the marginal likelihood, $p(D)$, expresses what the observations look like given the model. The posterior distribution indicates the uncertainty of the set of parameters after considering both the prior and the information of the data. The prior distribution expresses the belief of an uncertain quantity before considering the data or evidence. This belief is categorised into informative, non-informative and weakly informative prior. Informative expresses definite information about a variable , non-informative prior provides a small or general information about a variable and weak informative expresses partial information about a variable. The likelihood describes the connection of the data or evidence with the hypothesis. The marginal distribution is the total probability of observing all the data under all possible values of the unknown parameters. In other words, the Bayesian inference is a methodology that combines the empirical perception of a random process based on the observed data. The mathematical representation of the Bayes' theorem is

$$p(x|D) = \frac{p(D|x)p(x)}{p(D)} \tag{2.1}$$

where

$$p(D) = \int p(D|x)p(x)dx \tag{2.2}$$

When observing the data, the marginal likelihood is constant. Thus the posterior distribution is proportional to the prior multiplied by the likelihood. This constant is required to normalise the product of the likelihood and the prior probability density functions. The steps of the Bayesian inference include:

1. The definition of the likelihood function, $p(D|x)$.
2. The definition of the prior distribution, $p(x)$
3. The computation of the posterior distribution, $p(x|D)$, using Bayes' theorem.
4. Inference from the posterior distribution.

The marginal likelihood usually does not have a closed form as it is approximated. Thus, the posterior is approximated, and this can be achieved using MCMC methods. This generates samples to describe an approximation of the posterior distribution.

2.3 Markov chain Monte Carlo methods

MCMC methods are applied in Bayesian statistics to generate samples from a distribution. These samples can be used for various purposes (e.g. computing integrals, such as the one defined by $P(D)$, using Monte Carlo integration). Spanning a wide class of algorithms, MCMC methods generate samples from a probability distribution by constructing and simulating a Markov chain until convergence to an equilibrium distribution. MCMC methods are categorised into two methodologies. One is to build a Markov chain sequentially. When the chain converges, the generated samples represent the estimation of interest. The random walk Metropolis-Hastings (RWMH) or other variations of this algorithm belong to this class. The second methodology is to create samples based on a proposed density function and assign weights to each sample accordingly (i.e., importance sampling), and examples include the Transitional MCMC and SMC methods. Algorithms from both classes are analysed in the following subsections.

2.3.1 Metropolis-Hastings

The MH algorithm is a variation of the Metropolis algorithm proposed by Metropolis et al. in 1953 [69] used in situations where the target distribution (posterior distribution) is intractable and thus hard to analyse. The algorithm simulates a Markov chain by starting from a sample (initial position) and explore the space of interest until convergence to stationarity. The samples are generated from a proposal distribution, $q(x^*|x)$. The most common proposal distribution is the random walk, where the new sample is generated using the Gaussian distribution, with mean value the preceding sample and variance ϵM, where ϵ is the step size and M the preconditioning matrix. The initial samples are dependent on the first sample and removed from the overall generated samples at the end of the simulation as burn-in, as the initial sample can be in a region with low density. According to the proposal distribution, the algorithm proceeds by sequentially generating a single sample during every iteration. The sample generation is supported through an accept-reject mechanism, which is the acceptance probability that decides if the proposed generated sample will be accepted or rejected. The proposal distribution is a user-defined probability density function (Algorithm 1).

Algorithm 1 Metropolis-Hastings algorithm

```
 1: Initalise $x_1 \sim q(x_1)$
 2: for $i = 1 : N$ do
 3:     Propose $x^* \sim \mathcal{N}(x^*|\epsilon x_i, \epsilon M)$
 4:     Calculate the acceptance probability $a = min\{1, \frac{\pi(x^*)q(x_i|x^*)}{\pi(x_i)q(x^*|x_i)}\}$
 5:     Sample $r \sim [0,1]$ uniform
 6:     if $a < r$ then
 7:         Accept the proposal, $x_{i+1} = x^*$
 8:     else
 9:         Reject the proposal, $x_{i+1} = x_i$
10:     end if
11: end for
```

Different variants for the algorithm exist, depending on the choice of the proposal distribution. The Metropolis-adjusted Langevin algorithm (MALA) (Algorithm 2) is a special case of the MH algorithm where the proposed candidate sample (and corresponding proposal distribution) is generated via Langevin dynamics [88]. The algorithm generates samples from the target density, $\pi(x)$. For every iteration of the algorithm, a proposed sample is generated, which includes gradient steps with inject of Gaussian noise

$$x^* = x + \frac{\epsilon^2}{2}\nabla_x \log(\pi(x)) + z \tag{2.3}$$

where $z \sim \mathcal{N}(0, \epsilon^2 M)$ is the integration step size. The $\mathcal{N}(\mu, \sigma^2)$ denotes the Gaussian (normal) distribution with mean value μ and variance σ^2. The $X \sim \mathcal{N}(\mu, \sigma^2)$ denotes a random variable X distributed normally with mean μ and variance σ^2. The proposal density is

$$q(x^*|x) = \mathcal{N}(x^*|x + \frac{\epsilon^2}{2}\nabla_x \log(\pi(x)), \epsilon^2 M) \tag{2.4}$$

and the acceptance probability [35]

$$\min\{1, \frac{\pi(x^*)q(x|x^*)}{\pi(x)q(x^*|x)}\} \tag{2.5}$$

This acceptance probability is an important parameter with twofold importance. First, it is a debugging tool on the MH algorithm and, second, it adapts the step size based on the optimal acceptance rate [65]. The optimal acceptance rate in the MH algorithm depends on the proposal distribution. If the proposal is a random walk, then the optimal acceptance rate is 0.234, and if a MALA proposal, then the rate is 0.574 [98]. In both cases, the step size can be adapted by comparison of the current average and optimal acceptance rates.

The preconditioning matrix (or scalar for one-dimensional problems), denoted with M, needs to be selected carefully as it can increase or decrease the acceptance rate of the MH algorithm. Even in toy examples, not tuning the preconditioning matrix can lead to poor mixing (very small step sizes leading to slow convergence) [83]. Notice this proposal can be viewed in two ways. First, if we remove the gradient steps, then the proposal will be a random walk. The MALA requires $O(N^{\frac{1}{3}})$ steps to converge to the target density, while random walk requires $O(N)$, where N is the number of iterations [98]. Second, if we remove the Gaussian noise and set M to one, then the equation can be viewed as the standard gradient ascent optimisation algorithm [89].

Algorithm 2 Metropolis-adjusted Langevin algorithm

```
1: Initalise $x_1 \sim q(x_1)$
2: **for** $i = 1 : N$ **do**
3:     Propose $x^* \sim \mathcal{N}(x^*|x_i + \frac{\epsilon^2}{2}\nabla_{x_i}\log(\pi(x_i)), \epsilon^2 M)$
4:     Calculate the acceptance probability $a = \min\{1, \frac{\pi(x^*)q(x_i|x^*)}{\pi(x_i)q(x^*|x_i)}\}$
5:     Sample $r \sim [0, 1]$ uniform
6:     **if** $a < r$ **then**
7:         Accept the proposal, $x_{i+1} = x^*$
8:     **else**
9:         Reject the proposal, $x_{i+1} = x_i$
10:    **end if**
11: **end for**
```

2.3.2 Hamiltonian Monte Carlo

Hamiltonian (or Hybrid) Monte Carlo (HMC) [30] is a Markov chain Monte Carlo sampling algorithm which explore the target of interest more efficiently than the MH algorithm and with larger acceptance rates [76, 38, 23]. In the HMC the proposal is generated using the Hamiltonian function, $H(x, r) = U(x) + K(r)$, where $U(x)$, $K(r)$ denote the potential and kinetic energies, respectively. The physical interpretation of the Hamiltonian function is discussed in [76] as the following. Consider a puck with a given position, x and a momentum, r, which slides on over a frictionless surface of varying height. The potential energy of the puck is proportional to the height of the surface at the current position, x, and the kinetic energy is based on the momentum, r, and the mass, M of the puck. If the puck is moving towards a rising slope, then it will continue to slide (i.e., decreasing kinetic energy and increasing potential energy) until the kinetic energy becomes zero. At this point, the puck will then slide in the opposite direction (i.e., increasing kinetic energy and decreasing potential energy). The potential and kinetic energies are defined by

$$\begin{aligned} U(x) &= -\log(\pi(x)) \\ K(r) &= \frac{r^\top M^{-1} r}{2} \end{aligned} \tag{2.6}$$

where $\pi(.)$ is the target distribution. The mass, M, is a symmetric, positive-definite matrix, which is a typical diagonal and is often a scalar multiple of the identity matrix. Based on Equations 2.6, to propose samples, the HMC simulates the Hamiltonian dynamics as

$$\begin{aligned} \frac{dr}{dt} &= -\frac{\partial U}{\partial x} \\ \frac{dx}{dt} &= M^{-1} r \end{aligned} \tag{2.7}$$

The Hamiltonian Equations 2.7, describe a continuous simulation from a state at time t to a state at time $(t+t')$.The implementation of Algorithm 3 approximates the Hamiltonian equation based on time discretisation, which can be achieved using a modification of the Euler and leapfrog methods [76]. As the equations are discretised, error is introduced, and the MH algorithm is employed to accept or reject the proposed new state.

The HMC introducing an auxiliary momentum vector and implementing Hamiltonian dynamics avoids the random walk behaviour, so the potential energy function is the target density. This allows the algorithm to perform larger steps that are less correlated and converge to stationarity faster than the random walk algorithm. This discussion offers a summary of the HMC, and a detailed analysis is available in [76].

2.3.3 Transitional Markov chain Monte Carlo

The Transitional MCMC (TMCMC), proposed by J. Ching et al. [24], is a population-based MCMC in cooperation with an annealing scheme. Initially, N samples are drawn from the prior distribution. The algorithm proceeds by constructing and sampling multiple intermediate distributions, using N samples for each distribution, until the convergence to the posterior distribution according to

$$\underbrace{\pi(x_k^{(i)})}_{\text{posterior}} \propto \underbrace{\pi(D|x_k^{(i)})^{p_i}}_{\text{likelihood}} \underbrace{\pi(x_k^{(i)})}_{\text{prior}} \tag{2.8}$$

where $i = 1, \ldots, m$ with $0 = p_0 < p_1 < \ldots < p_i < \ldots < p_m = 1$ and $k = 1, \ldots, N$ denote the TMCMC stage or iteration and the samples, respectively. In general, the

Algorithm 3 Hamiltonian Monte Carlo

```
 1: Initialise the position $x_0$ and the step size $\epsilon$
 2: for $t = 1 : N$ do
 3:     Compute the momentum $r \sim \mathcal{N}(0, M)$
 4:     Compute the Energy function $H_1 = \frac{r^\top M r}{2} - log(\pi(x_t))$
 5:     Leapfrog Integration for $L_s$ steps
 6:     $r = r - \frac{\epsilon}{2}\frac{\partial \pi(x)}{\partial x}$
 7:     for $j = 1 : L_s$ do
 8:         $x = x + \epsilon \frac{r}{M}$
 9:         $r = r - \frac{\epsilon}{2}\frac{\partial \pi(x)}{\partial x}$
10:     end for
11:     Compute the new Energy function $H_2 = \frac{r^\top M r}{2} - \log(\pi(x_t))$
12:     Correction via the Metropolis-Hastings
13:     $r_u \sim U[0, 1]$ uniform
14:     if $\exp(H_2 - H_1) < r_u$ then
15:         Reject: $x_t = x_{t-1}$
16:     else
17:         $x_t = x$
18:     end if
19: end for
```

choice of intermediate distributions, which is managed from the exponent p_i, should be slow enough to guarantee the desirable smooth transition from the (i) distribution to the $(i + 1)$. In every TMCMC iteration, a weight is assigned for each sample by

$$w(x_k^{(i)}) = \pi(D|x_k^{(i)})^{p_{i+1}-p_i} \tag{2.9}$$

The intermediate, p values, are selected so that the coefficient of variation, or relative standard deviation, of the weights is equal to 100%. To obtain the samples $x_j^{(i+1)} = x_j^i$ from the $\pi(x_k^{(i+1)})$, resampling is performed with the probability on the normalised weights as

$$\tilde{w}_k^{(i)} = \frac{w_k^{(i)}}{\sum_{j=1}^{N} w_k^{(j)}} \tag{2.10}$$

The resampling step is mandatory to avoid the degeneracy phenomenon [12]. Also, the resampling algorithm eliminates the low weighted samples and replicates them with larger weighted samples. As a result, only a few Markov chains will grow during the m iterations, which is undesirable as the TMCMC algorithm is initialised to use N distinct Markov chains. The solution proposed in [24] applies the MH algorithm for every resampling, and the proposal distribution is centred at the preceding sample with covariance

$$\Sigma_i = \beta^2 \sum_{k=1}^{N} \tilde{w}_k^{(i)} \big[x_k^{(i)} - \sum_{j=1}^{N}(x_j^{(i)}\tilde{w}_j^{(i)})\big]\big[x_k^{(i)} - \sum_{j=1}^{N}(x_j^{(i)}\tilde{w}_j^{(i)})\big]^{\mathsf{T}} \tag{2.11}$$

where β is a scaling parameter used to control the rejection rate and it is recommended [24] to be equal with 0.2.

Improved algorithmic versions of the TMCMC (Algorithm 4 or basic algorithm in [24]) exist, which lead to more efficient exploration of the space compared to the original. In [11], the proposal distribution is based on the Langevin forward kernel. In [19], an adapted annealing scheme and burn-in for the sampling procedure in every TMCMC iteration are proposed.

Algorithm 4 Use TMCMC to simulate $\theta_{k=1:N}^{(i)}$

```
 1: $i = 1$
 2: Sample $\theta_k^{(0)} \sim \pi(x)$
 3: Initialise the scaling parameter $\beta = 0.2$ and the exponent $p_1 = 0$
 4: while $p_i < 1$ do
 5:     $p_l = p$; $p_h = 2$;
 6:     while $p_h - p_l > 10^{-6}$ do
 7:         $p_{tmp} = \frac{p_l + p_h}{2}$
 8:         Compute the sample weights $w(\theta_k^{(i)}) = \pi(D|\theta_k^{(i)})^{p_{i+1}-p_i}$
 9:         Compute the coefficient of variation $cv_w = \frac{\sigma_w}{\overline{w}} = \frac{\text{standard deviation}}{\text{mean}}$
10:         if $cv_w > 1.1$ then
11:             $p_h = p_i$
12:         else
13:             $p_l = p_i$
14:         end if
15:     end while
16:     $p_i = p_{tmp}$
17:     if $p_i > 1$ then
18:         break
19:     end if
20:     Compute the normalised weights $\tilde{w}_k^{(i)} = \frac{w_k^{(i)}}{\sum_{j=1}^{N} w_k^{(j)}}$
21:     Use the normalised weights to resample and generate the new set of samples.
22:     Use the Metropolis algorithm for each new set of sample to compute $\theta_k^{(i+1)}$ using
        the Gaussian distribution as proposal centered at the preceding ("lead") sample
        using Equation 2.11 as covariance
23:     $i = i + 1$
24: end while
```

2.4 Sequential Monte Carlo methods

Sequential Monte Carlo (SMC) samplers and Particle Filters belong to a wider class of methods called Sequential Monte Carlo (SMC) methods and perform statistical inference. The SMC sampler generalises Particle Filters as it is applicable in both static and dynamic distributions of interest [66]. Both methods are applicable in systems with non-linear and non-Gaussian characteristics. In this section, a brief description of the two methods is provided with reference to [12] and [72] for detailed descriptions.

2.4.1 Sequential Monte Carlo samplers

2.4.1.1 Initialisation and posterior distribution

In SMC samplers the target distribution is constructed by using a sequence of artificial distributions $\pi_1(x_1)$, $\pi_2(x_2)$, $\dots$, $\pi_k(x_k)$, where $\pi_k(x_k)$ is the final target distribution (distribution of interest). The algorithm begins by drawing N particles from an initial importance probability density function using the standard importance weights with proposal distribution $q_1(x_1)$

$$w_1(x_1) = \frac{\pi_1(x_1)}{q_1(x_1)} \tag{2.12}$$

Beyond the first iteration, the cloud of particles are propagated by using the sequence of artificial targets as a sequence of backward Markov kernels, $L_k(x_{k-1}|x_k)$, as

$$\pi(x_{1:k}) = \pi_k(x_k) \prod_{k=2}^{k} L_k(x_{k-1}|x_k) \tag{2.13}$$

2.4.1.2 Proposal distribution and importance weights

Given a set of weighted particles that approximate the $k-1$ artificial target, the next artificial target, k, is approximated by sampling from the forward Markov kernel, $q_k(x_k|x_{k-1})$ such that

$$q(x_{1:k}) = q(x_1) \prod_{k=2}^{k} q(x_k|x_{k-1}) \tag{2.14}$$

Each particle is associated with a weight by

$$w_k(x_{1:k}) = w_{k-1}(x_{1:k}) \frac{\pi_k(x_k)}{\pi_k(x_{k-1})} \frac{L_k(x_{k-1}|x_k)}{q_k(x_k|x_{k-1})} \tag{2.15}$$

To avoid numerical issues, Equation 2.15 is expressed using a logarithmic scale, and the importance weights are normalised as

$$\tilde{w}_k^{(i)} = \frac{w_k^{(i)}}{\sum_{j=1}^{N}(w_k{}^{(j)})} \tag{2.16}$$

where $\sum(\tilde{w}_k) = 1$, $\tilde{w}_k \in \mathbb{R}^{N\times 1}$ and $i = 1, \ldots, N$ represent the particle index with N total number of particles.

2.4.1.3 Degeneracy phenomenon and effective sample size

Similarly, with Particle Filters, the weighted particles may be resampled after the importance weights evaluation. This resampling step reduces the variability of the importance weights (degeneracy phenomenon) as the negligible particles are eliminated and substituted with more important particles. Resampling is triggered according to the effective sample size [12]

$$N_{eff} = \frac{1}{\sum_{i=1}^{N}(\tilde{w}_k^{(i)})^2} \tag{2.17}$$

where $N_{eff} \in [1, N]$ and $i = 1, \ldots, N$ denotes the particle index.

2.4.1.4 SMC samplers emulate MCMC

The SMC Sampler is an alternative method to Markov chain Monte Carlo (MCMC) methods (e.g., the Metropolis-Hastings algorithm). The user-defined backward Markov kernel can be selected to emulate MCMC as $L(x_{k-1}|x_k) = q_k(x_k|x_{k-1})$. Further discussion on the backward Markov kernel is included in Chapter 7.

2.4.1.5 Estimation

In the basic SMC Sampler, estimations are performed according to the particles in the final iteration. The expected value is computed by multiplication of the final particles with the corresponding weights

$$f = \sum_{i=1}^{N} x_K^{(i)} \tilde{w}_K^{(i)} \tag{2.18}$$

In this approach only the particles, denoted with i, of the last iteration, denoted with K, are considered for the final estimation. In Chapter 4, this process is extended and includes an available method as well as a novel method where the final estimation is computed using the particles during all iterations.

Algorithm 5 Basic SMC sampler

```
1:  for $i = 1 : N$ do
2:      Sample $x_1^{(i)} \sim q(x_1^{(i)})$
3:      Calculate $w_1^{(i)} = \frac{\pi(x_1^{(i)})}{q(x_1^{(i)})}$
4:  end for
5:  for $k = 2 : K$ do
6:      for $i = 1 : N$ do
7:          Sample $x_k^{(i)} \sim q(x_k^{(i)}|x_{k-1}^{(i)})$
8:          Calculate $w_k^{(i)} = w_{k-1}^{(i)} \frac{\pi(x_k^{(i)})L(x_{k-1}^{(i)}|x_k^{(i)})}{\pi(x_{k-1}^{(i)})q(x_k^{(i)}|x_{k-1}^{(i)})}$
9:      end for
10:     Weights Normalisation $\tilde{w}_k = \frac{w_k}{\sum(w_k)}$
11:     Calculate the effective sample size $N_{eff} = \frac{1}{\sum_{i=1}^{N}(\tilde{w}_k^{(i)})^2}$
12:     if $N_{eff} < N_T$ then
13:         for $i = 1 : N$ do
14:             Resampling (Alg. 7) with $\tilde{w}_t$, to produce the new population, $x_t$
15:             Set $w_k^{(i)} = \frac{1}{N}$
16:         end for
17:     end if
18: end for
```

2.4.2 Particle filters

A range of different Particle Filter methods exist, and this section provides a brief description of the GENERIC particle filter, while a detailed analysis for this and other methods is available in [12].

2.4.2.1 Sequential importance resampling

Particle filters assume a dynamic stream of data, where the current state, x_t, is a sufficient estimation of the history of the states $x_{1:t-1}$ [66]. Consider a time evolving distribution, $\pi(x_t)$ at time t with state transition

$$x_t|x_{t-1} \sim \pi(x_t|x_{t-1}) \tag{2.19}$$

with an initial distribution (or prior), $\pi(x_0)$, and an incoming stream of measurements (or observations)

$$y_t|x_t \sim \pi(y_t|x_t) \tag{2.20}$$

The approximation of the posterior distribution [12]

Algorithm 6 GENERIC Particle filter

```
 1: Sample $x_0 \sim \pi(x_0)$
 2: Assign $w_0 = \frac{1}{N}$
 3: for $t = 1 : T$ do
 4:     for $i = 1 : N$ do
 5:         Sample $x_t^{(i)} \sim q(x_k^{(i)}|x_{k-1}^{(i)}, y_t^{(i)})$
 6:         Calculate $w_t^{(i)} = w_{t-1}^{(i)} \frac{\pi(x_t^{(i)}|x_{t-1}^{(i)})\pi(y_t^{(i)}|x_t^{(i)})}{q(x_t^{(i)}|x_{t-1}^{(i)},y_t^{(i)})}$
 7:     end for
 8:     Weights Normalisation, $\tilde{w}_t = \frac{w_t}{\sum(w_t)}$
 9:     Calculate the effective sample size, $N_{eff}$ (Eq. 2.17)
10:     if $N_{eff} < N_T$ then
11:         for $i = 1 : N$ do
12:             Resampling (Alg. 7) with $\tilde{w}_t$, to produce the new population, $x_t$
13:             Set $w_t^{(i)} = \frac{1}{N}$
14:         end for
15:     end if
16: end for
```

Algorithm 7 Minimum Variance Resampling

```
Input: $x_t, w_t, N$
Output: $x_t, w_t$
 1: $ncopies = \texttt{MVR}(w_t)$ [47]
 2: $(ncopies, x_t) = \texttt{quickSort}(N, ncopies, x_t)$
 3: $x_t = \texttt{Redistribute}(N, ncopies, x_t)$
```

$$\pi(x_t|y_t) = \frac{\pi(y_t|x_t)\pi(x_t|x_{t-1})}{q(y_t|y_{t-1})} \tag{2.21}$$

$$\propto \pi(y_t|x_t)\pi(x_t|x_{t-1}) \tag{2.22}$$

is achieved by using a set of weighted particles. Importance weights are computed, similarly to SMC Samplers, so that each particle is assigned to a weight [12]

$$w_t = w_{t-1}\frac{\pi(x_t|x_{t-1})\pi(y_t|x_t)}{q(x_t|x_{t-1}, y_t)} \tag{2.23}$$

In the SIS step, the particles are propagated according to sampling from the proposal, $q(x_t|x_{t-1}, y_t)$, followed by the weights assignment for each particle. Resampling, equivalently to SMC Samplers, is triggered according to the effective sample size. Algorithm 6 shows the pseudocode for the GENERIC Particle Filter. The algorithm relies on several functions, which are covered in detail in Chapter 4. Briefly these functions include:

- Importance sampling using the proposal distribution, $q(x_t|x_{t-1}, y_t)$

- $ncopies = \texttt{MVR}(w_t)$, where MVR stands for Minimum Variance Resampling and determines the number of times each particle needs to be replicated. The function takes the particles' weights at the current time t, w_t, as input.

- $(ncopies, x_t) = \texttt{quickSort}(N, ncopies, x_t)$ calculates the permutation that would sort vector $ncopies$, and applies this permutation to both inputs. While this sort is not necessary with a single processor implementation, in Chapter 4 we will exploit the fact that the output is sorted.

- $x_t = \texttt{Redistribute}(N, ncopies, x_t)$ returns the new population of particles given the old population and the number of replication each particle requires.

2.5 Sampling experiments

An initial comparison of the three Markov chain Monte Carlo (MCMC) methods and the basic Sequential Monte Carlo sampler is discussed in this section. All methods estimate the true mean value of the posterior distribution using 10000 samples and are evaluated using the last log mean squared error (MSE) provided in Table 2.1. In Figure 2.1 , the performance is expressed using the log mean squared error as well as the percentage of computations for the MALA, HMC, TMCMC, and MH, which implies the same total number of samples for each method. The last iteration log MSE describes the best estimate each algorithm achieves.

The HMC method outperforms all other methods when the step size is appropriately defined as its performance is sensitive to this value. The MALA and TMCMC have similar performance, and the basic SMC Sampler and MH methods result in the worst performance.

It is expected the basic SMC Sampler has a worse accuracy compared to the MALA and TMCMC as the former method's proposal distribution is based on the Euler discretisation, while the latter uses annealing. However, it is surprising that it is difficult to outperform or have similar accuracy as the MH algorithm since both methods use the same proposal distribution. A similar result is discussed in Chapter 3.

Further comparisons with these methods are discussed in the final three chapters where our effort is concentrated on improving the accuracy of the SMC sampler.

Table 2.1: Last iteration log mean squared error (MSE) of the MALA, TMCMC, MH, basic SMC sampler and HMC. Each method generates 10000 samples to estimate the true mean value.

Method	**Log MSE**
MALA	-7.98
MH	-7.19
TMCMC	-7.83
basic SMC sampler	-6.01
HMC	-11.63

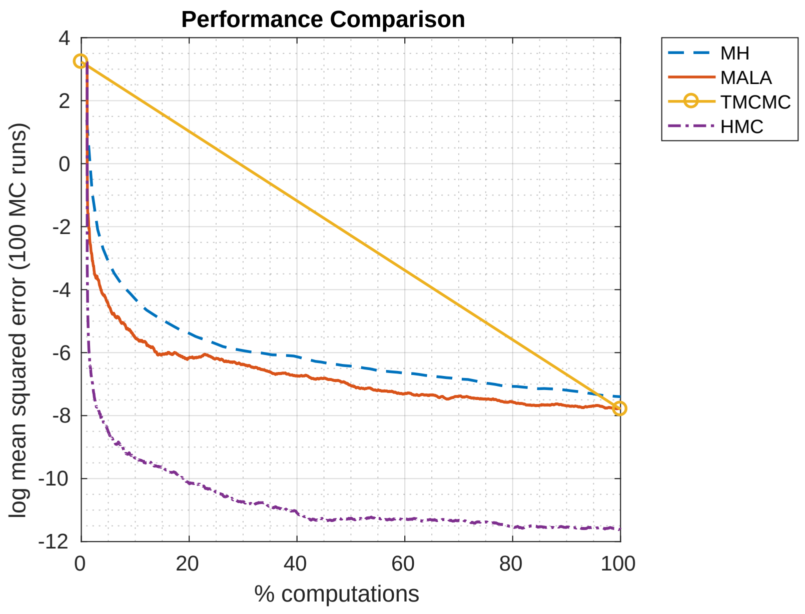


Figure 2.1: Comparison of MALA, HMC, MH and TMCMC on the Gaussian distribution, $\mathcal{N}(0, 1)$. Each method generates 10000 samples to estimate the true mean value.

2.6 Conclusions

In this chapter, we provided the description of a variety of MCMC methods and two SMC methods. For an initial comparison, we noticed that the HMC outperforms all other methods, but it is challenging to tune the user-defined parameters [76]. Methods for tuning those parameters exist, but the focus of the thesis examines and prioritises Langevin-based proposals, while the core method is the SMC Sampler. Further analysis and research are discussed in Chapters 6 and 7.

In Chapter 3, background on two widely-known deep learning methods is discussed, and a new method is proposed to train the Radial Basis Function (RBF) network using steps of importance sampling and resampling (i.e., the core methods in any SMC method) as the training method. Comparison of the proposed method with the original algorithm (i.e., the MH algorithm) reveals similar behaviour in the results with the sampling experiment from this chapter.

Chapter 3

Background on traditional deep learning algorithms and a new method to train the Radial Basis Function network

3.1 Introduction

Machine learning is the science of getting computers to act without being explicitly programmed [1]. Artificial neural networks (or neural networks) [40] and deep learning [55] are two machine learning tools consisting of algorithms or networks to detect features from a given dataset for the discovery of patterns or perform a defined task. The algorithms are programmed for automatic training. These training or learning procedures are categorised as supervised, unsupervised or semi-supervised.

In supervised learning (or learning with a teacher), the algorithm is provided with a dataset that includes the correct answers. This dataset uses labelled data in the form (x, d), where x is the input and d is the corresponding correct answer for the given x. In unsupervised learning (or learning without a teacher) we give the algorithm unlabelled data (without providing the "right" answer) x, and the system is trying to classify the given data. Other learning techniques exist such as the semi-supervised learning, a "hybrid" approach where the dataset is partially labelled.

Many applications are based on regression, classification and other tasks [39]. An example of a regression problem is to predict the price of new real estate given a dataset containing related previous values [82]. A classification task considers grouping or classifying the given inputs, such as categorising the genre of art films [93].

This chapter begins in Section 3.2 with a background of the learning procedures for two widely-applicable deep learning algorithms, the stacked autoencoder and deep belief network. In Section 3.3, a new method is proposed to train the RBF network using, as its core method, steps of importance sampling and resampling instead of the MH algorithm. Examples are included in each section, while Section 3.4 draws conclusions.

3.2 Background on deterministic and stochastic deep learning methods

This section reviews two traditional deep learning algorithms, the stacked autoencoder and the deep belief network. The stacked autoencoder is a deterministic network, and the deep belief network is stochastic. Descriptions of the training processes for a deterministic neural network, single layer perceptron, and multiple layer perceptron are provided in [40].

A Single Layer Neural Networks also known as Single Layer Perceptron (SLP) [40] is a neural network with two layers, an input layer and an output layer. The input layer units are fully connected with the output layer units. The output layer units or activation units perform a mathematical operation. A bias is used to shift the activation function horizontally (left or right). The default value of the bias unit is 1. The procedure of calculating the output of the activation unit or units is called forward propagation. The most common function used by the activation unit is the sigmoid : $f(x) = \frac{1}{1+\exp{(-x)}}$, where $f(x) \in [0, 1]$. Another example of activation unit is the hyperbolic tangent, while a list of different choices is available in [40].

Multi Layer Neural Networks also known as Multi Layer Perceptrons (MLP) use more that two layers, the input layer, the hidden layer or layers and the output layer. The capacity of neural networks describe the type of problem the network can solve. A single neuron can solve linear separable problems. This is a line that can separate the two classes (e.g. AND-Boolean operator, OR-Boolean operator). A neural network with a single hidden layer can solve universal approximations [48].

Neural Networks have the innovative ability to get trained and learn (or gain knowledge) through the training procedure. Mathematically this is achieved via the properly adjustment of weights and biases of the neural network in order to minimise the predefined cost function. The cost function defines the difference between the network output with the real target. There are many optimisation algorithms which can be used to minimise the cost function. A widely used approach on solving this problem is the back propagation algorithm [40] in conjunction with gradient descent as the optimisation method.

3.2.1 The stacked autoencoder

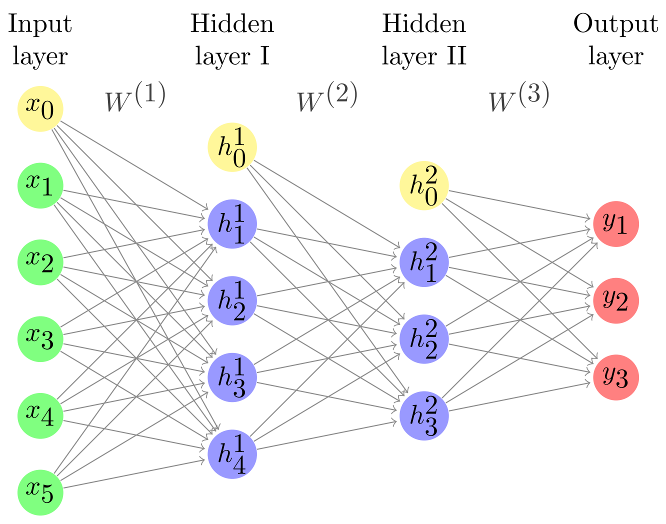


FIGURE 3.1: Stacked Autoencoder Network

The stacked autoencoder [77] [102] is a deep learning network consisting of multiple layers of autoencoders. An autoencoder is an unsupervised learning algorithm with the training objective to reconstruct a given input [77]. The network consists of an equal number of neurons in the input and output layers, while an intermediate hidden layer exists with either a smaller or larger number of neuron units. As mentioned, the basic component of the stacked autoencoder is a single autoencoder. The training of this one autoencoder follows the traditional approach of training a multiple layer neural network. The learning procedure is unsupervised, which means that the correction of the weights and biases on the network is achieved through the adjustment of each value according to a cost function describing the difference between the given input and current output. Training a stacked autoencoder is achieved through five stages:

1. Design the architecture of the network. This is related, for example, with the number of layers or autoencoders the network contains.

2. Train the first autoencoder. After the training is completed, forward propagate the first autoencoder features as an input to the second autoencoder. These features are the result of the multiplication of the trained first autoencoder weights and biases with the activation functions in the hidden layer.

3. Train the second autoencoder. Forward propagate the features of the trained second autoencoder as an input to the third autoencoder.

4. Continue the above procedure until the last autoencoder of the stacked autoencoder is trained.

5. In the final stage, fine-tune the stacked autoencoder network for better results. Here, back-propagation is used to improve the performance (i.e., adjusting the weights and biases of the overall network) of the autoencoders. This process is mandatory if we want to achieve better accuracy in the results.

3.2.2 The Deep Belief Network

3.2.2.1 Restricted Boltzmann machine

The Boltzmann machine is a stochastic network introduced in 1985 by G.E. Hinton [9] that combines statistical mechanics and neural networks. The network is undirected and fully connected, and the energy of the model corresponds to the network configuration (i.e., all connections of the network including the biases of the hidden and visible neurons).

$$E(v) = E(v; w) = -\sum_{i<j} s_i s_j w_{ij} + \sum s_i b_i \tag{3.1}$$

where v denotes the state vector (the data), s_i denote the binary state assigned to unit i, b_i the bias assigned to unit i and the w_{ij} the weight connection between units i and j. Probabilities are assigned to every possible state with

$$p(v) = \frac{e^{-E(v)}}{\sum_v e^{-E(v)}} \tag{3.2}$$

The aim of the training procedure is to find weights and biases that define a Boltzmann distribution in which the training vectors have high probability [44]. Each state vector, v, persists long enough for the network to reach thermal equilibrium [40]. This is achieved by differentiation of Equation 3.1 and using the fact that $\frac{\partial E(v)}{\partial w_{ij}} = -s_i s_j$

$$\sum_{v \in data} \frac{\partial log(p(v))}{\partial w_{ij}} = \langle s_i s_j \rangle_v - \langle s_i s_j \rangle_{model} \tag{3.3}$$

where the $\langle s_i s_j \rangle_v$ is the expected value of product of states $s_i s_j$ in the data distribution (i.e. positive phase) and $\langle s_i s_j \rangle_{model}$ is the expected value when the Boltzmann machine is sampling state vectors from its thermal equilibrium (i.e. negative phase). This difference of correlations is applied to update the weights of the network

$$\Delta w_{ij} \propto \langle s_i s_j \rangle_v - \langle s_i s_j \rangle_{model} \tag{3.4}$$

This model is impractical because the training procedure requires a very long time and real-world applications require many neurons.

During the following years, research efforts focused on simplifying the Boltzmann machine or proposing closely-related networks. In 1992, R. Neal [75] proposed the sigmoid belief network, which is a directed graphical model similar to a multilayer perceptron. The network does not require the negative phase during the computation of the derivative of the log-likelihood of the visible units. The negative phase in the Boltzmann machine is used for the computation of the global partition function with a mathematical description provided for the restricted Boltzmann machine [75]. As a result, the sigmoid belief network reduces the computation complexity and the required time for the network to reach thermal equilibrium. The problem with this approach is a phenomenon called "explaining away" [45], which implies that the independent latent variables become dependent when they influence an observable unit. Other efforts, alternative to MCMC, focus on using different approaches, such as variational methods, which is also used in deep sigmoid belief networks [71]). The goal is to maximise a lower bound of the log-likelihood of the visible units, while the training is achieved via the expectation-maximisation (EM) algorithm. The convergence is faster than MCMC methods but sacrifices the accuracy of the model [45].

RBM's are special cases of the Boltzmann machines where there is no connection between the units of the same layer, but only between the visible and hidden layers [17]. Assuming a restricted Boltzmann machine with m visible and n hidden units (Figure 3.2), the joint configuration, (v,h) of the model is given by the energy function

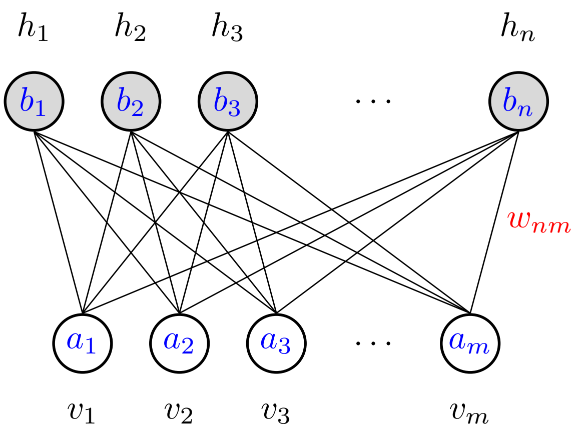


FIGURE 3.2: Model representation of the restricted Boltzmann machine with m visible and n hidden units.

$$E(v,h) = E(v,h;w) = -h^{\intercal}Wv - a^{\intercal}v - b^{\intercal} \tag{3.5}$$

where the weights matrix, w, represents the connections between the visible and the hidden units, a and b are the biases of the visible and the hidden units, and $v \in \{0,1\}^m$ and $h \in \{0,1\}^n$ denote the states of the visible and hidden units, respectively. Originally, the RBM as well as the Boltzmann machine use binary units, however this is not mandatory [33]. Probabilities are assigned to every possible visible and hidden unit with

$$p(v,h;w) = \frac{e^{-E(v,h)}}{Z} \tag{3.6}$$

where Z is the normalising constant or partition function

$$Z = \sum_{v,h} e^{-E(v,h)}$$

The probability that the model assigns to the visible vector v is

$$p(v; w) = \frac{\sum_h e^{-E(v,h)}}{Z} = \frac{\sum_h e^{-E(v,h)}}{\sum_{v,h} e^{-E(v,h)}} \tag{3.7}$$

To train an RBM (as discussed in [26] the training process can lead to poor local solutions) the maximisation of the log-likelihood of the $p(v)$ is required. Numerically, this implies finding where the derivative of the log-likelihood with respect to the weights is equal with zero such that

$$\begin{aligned}
\frac{\partial \log p(v; w)}{\partial w} &= 0 \Leftrightarrow \\
\frac{\partial \log \frac{\sum_h e^{-E(v,h)}}{Z}}{\partial w} &= 0 \Leftrightarrow \\
\frac{\partial \log \sum_h e^{-E(v,h)}}{\partial w} - \frac{\partial \log \sum_{v,h} e^{-E(v,h)}}{\partial w} &= 0 \Rightarrow \\
\underbrace{v_i \cdot p(h_j = 1|v)}_{\text{positive phase}} - \underbrace{p(v_i = 1, h_j = 1)}_{\text{negative phase}} &= 0
\end{aligned} \tag{3.8}$$

The derivative of the log-likelihood is the difference between the data-dependent and the model expectations, known as the positive and negative phases, respectively. The positive phase increases the probability of the data by reducing the energy, while the negative phase reduces the probability of the samples generated by the model by increasing the energy. The conditional distributions over the visible and the hidden units are given by

$$p(v_i = 1|h) = \sigma(a_i + h^{\intercal} w_i) \tag{3.9}$$

$$p(h_j = 1|v) = \sigma(b_j + w_j v) \tag{3.10}$$

where $i \in \{1, ..., m\}$, $j \in \{1, ..., n\}$, w_i and w_j are the i^{th} row and j^{th} column of the weights, $w_{ji} \in \mathbb{R}^{n \times m}$, respectively. The σ denotes the sigmoid function

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.11}$$

The positive phase is easily computed from the conditional distributions. The negative phase is intractable because an exponential summation over both the visible and the hidden units is required [26]. Gradient ascent with learning rate, η, can be applied to update the weights of the system with

$$\Delta w = \eta \cdot \{v_i \cdot p(h_j = 1|v) - p(v_i = 1, h_j = 1)\} \tag{3.12}$$

The idea of the contrastive divergence method is to apply the Gibbs sampling to compute the negative phase of the log-likelihood gradient (Equation 3.5) by providing the current training example to the visible layer as the initial value of the Gibbs sampler. The next steps of the algorithm consist of running in turns the conditional distributions provided in Equations 3.6 and 3.7.

3.2.2.2 Deep Belief Network

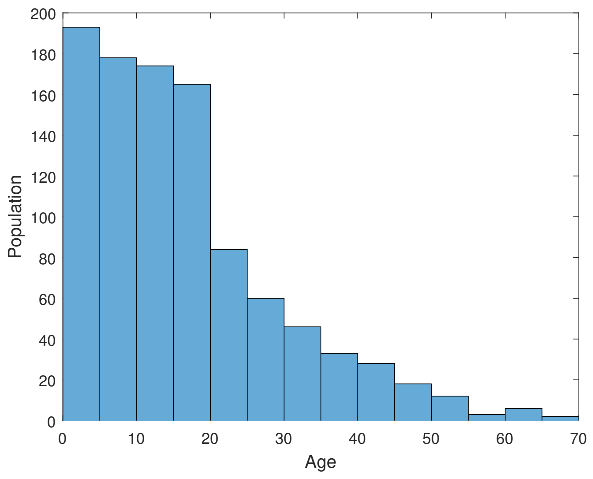
The strategy for training a deep belief network is similar to the stacked autoencoder. The basic component of a deep belief network is the restricted Boltzmann machine (RBM). The general proposed method for training deep learning algorithms, as discussed in the previous section, is to use a basic component trained first with a grid-layer pre-training procedure followed by a fine-tuning step. This latter step is not mandatory but leads to better performance. During the first step, a stack of the basic component is created where each component is trained, while the second step views the deep network as a single network and adjusts all weights and biases (e.g., backpropagation in the entire network) [45].

3.2.3 Face age classification

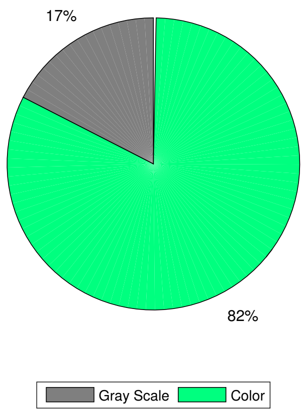
Faces provide a significant source of information, such as the age, gender, expression, and ethnicity. There are many applications related to face age estimation, such as security control and human-computer interaction. To demonstrate the two-deep learning algorithms discussed above, the benchmark Face and Gesture Recognition Research Network (FG-Net) ageing database [80] is pre-processed and applied to the deep belief network and the standard stacked autoencoder.

3.2.3.1 Preprocessing the FG-Net aging database

The FG-Net ageing database contains 1002 face images from 82 individuals with ages ranging between new-born to 69 years [80]. The 72% (730) of which are new-born to 20 years (Figure 3.3). The images are separated based on the human growth curve. From the 1002 images, 175 are grayscale, and the remaining 827 are colour images. Each image in the dataset (with one exception) is annotated with 68 landmark points located at key positions.



(a) Age Histogram



(b) Number of grayscale images

Figure 3.3: (a) Age histogram and (b) number of grayscale and color images in the FG-NET aging database (right).

The data pre-processing is a very critical step in machine learning applications. It allows the deep structure to detect or extract meaningful features while reducing misleading results based on the input. The pre-processing for FG-Net contains the grayscale normalisation, face alignment, cropping, and resizing. Without the included landmarks, the equivalent pre-processing procedure would require more advanced methods (e.g., face and eyes localisation). Figure 3.5 a sample of the pre-processed images.

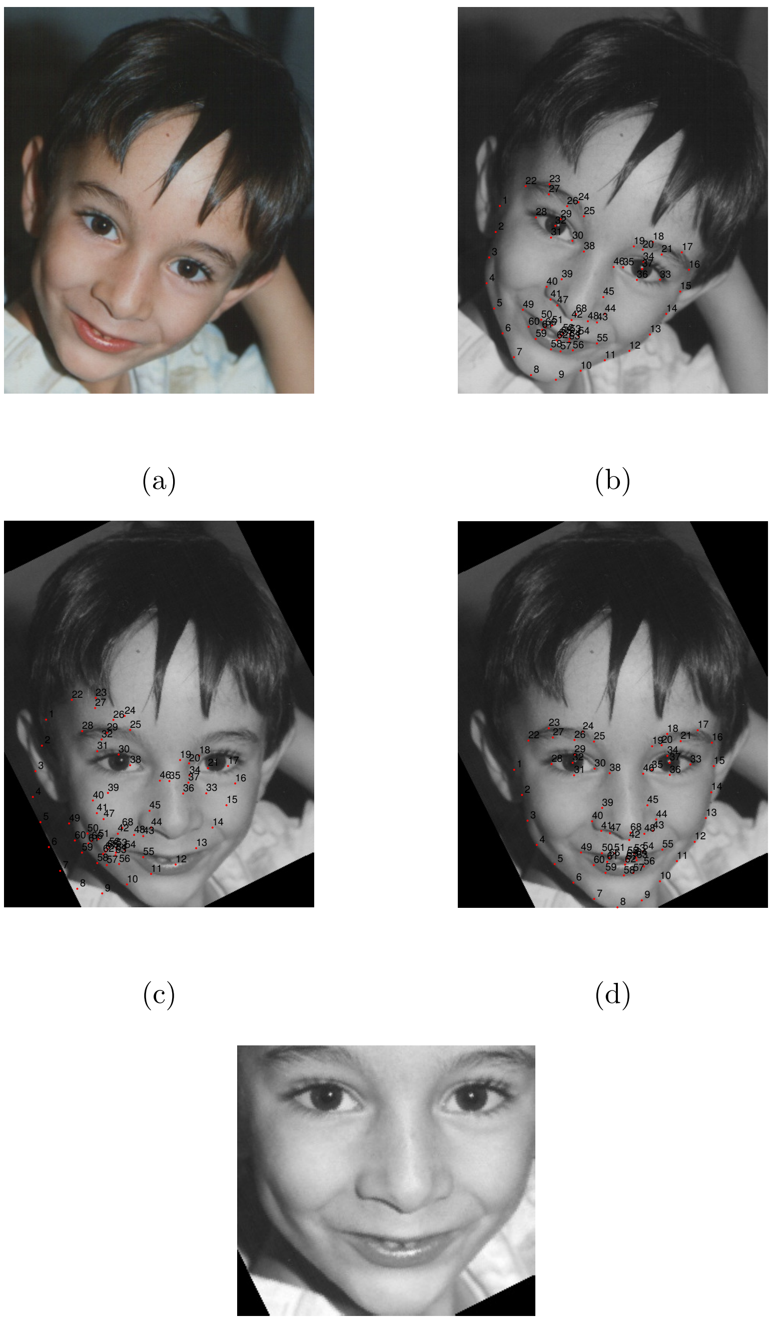
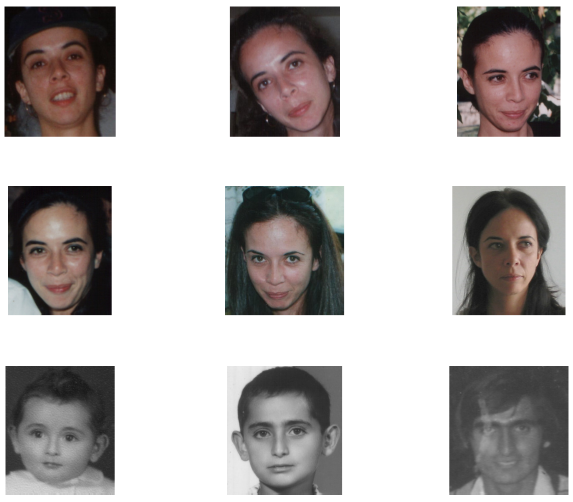
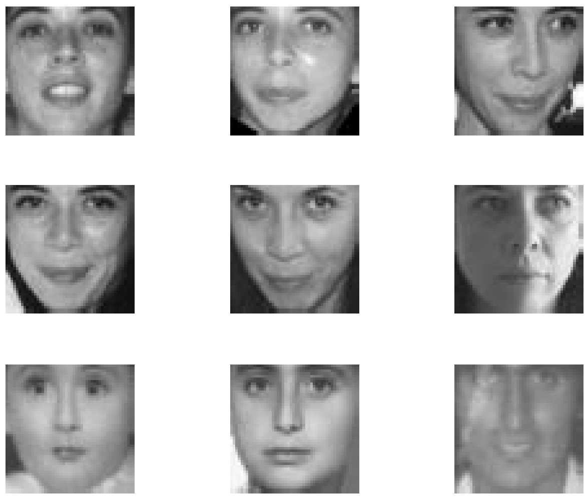


FIGURE 3.4: The initial image (a) is converted to grayscale (b). The image is rotated, but the landmarks are not changed yet (c). The new position of the landmarks are computed based on the rotation matrix (d). The final image (e) is cropped based on the landmarks.



(a) Sample of initial images



(b) Preprocessed Images

FIGURE 3.5: (a) A sample of the initial images and (b) the corresponding preprocessed images (right).

3.2.3.2 FG-Net aging database classification with deep learning

Our benchmark is evaluated with the standard stacked autoencoder (deterministic network) and the deep belief network (stochastic network) deep learning algorithms.

In the example with the stack autoencoder, two autoencoders (784-100-50 number of neurons for each layer) are considered followed by a softmax classifier [77]. In a classification task, the "decoding" layers of the stacked autoencoder are removed, and the features of the last autoencoder are connected to a softmax classifier. The procedure begins with the training of the first autoencoder, and when completed, the features or filters learned during the training are forward propagated to the second layer. This procedure is followed by the next autoencoder with the difference in that the second layer propagates to the softmax classifier. Finally, the fine-tuning procedure views the deep structure as a single network and uses forward and back propagation to update the weights and biases of the entire network as a standard multiple layer perceptron (MLP).

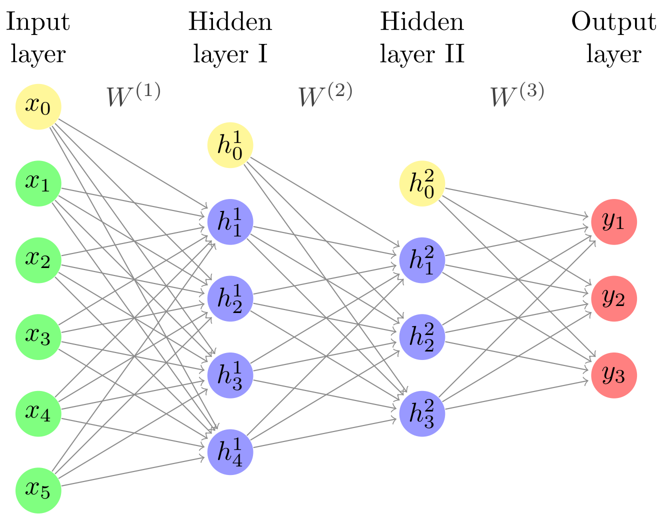
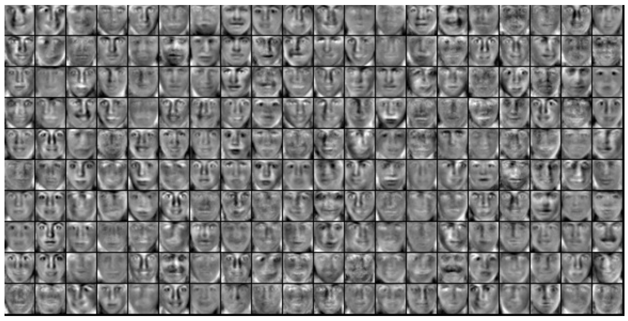


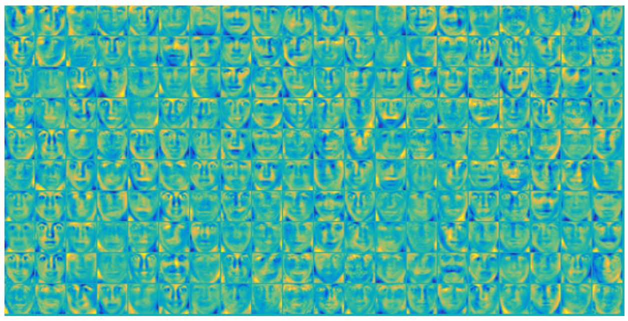
FIGURE 3.6: Example of the stacked autoencoder network

For every step of the algorithm, the standard gradient descent is used to train the network. Specifically, the "minFunc" library for the gradient descent is used, which is the Limited memory-Broyden Fletcher Goldfarb Shanno (L-BFGS) optimisation algorithm [58]. The final accuracy is heavily based on the number of classes and the size of the training and testing sets. We performed two examples with the learning (or training) procedure supported with 900 images, while the testing procedure with the remaining from the set. The classification task groups the faces according to age using a classifier with six classes: 0-5, 6-10, 11-15, 16-20, 21-31, and 31-69. The performance achieved from the network is $\sim 48\%$. If the initial configuration includes classes with larger age range values, then the performance improves. For example, if the goal is to group the faces using age ranges $0-10, 11-20, \ldots, 61-70$, then the accuracy is $\sim 73\%$. Finally, the first layer filters learned is provided in Figure 3.7.

The deep belief network is applied in a three layer (784-500-500 number of neurons for each layer) of RBM's and 6 classes 0-5, 6-11, ..., 31+. The performance is slightly better than the stacked autoencoder, $\sim$ 51%, which is compared with a k-step contrastive divergence with $k = 5$ and 10 but without significant difference. It is known that tuning an RBM and a deep algorithm are challenging while debugging is usually done through visualisation [106]. It is interesting that the filters the algorithm learned for the first RBM of the deep belief network after the training procedure are not satisfactory (Figure 3.8). In contrast, the stacked autoencoder filters of the first layer appear better (Figure 3.7). The histogram of the filters (Figure 3.9) confirms the mean absolute magnitude of the filters increases by a factor of 10^2 [106]. However, this does not suggest any sign of overfitting or underfitting the dataset. Similarly, the error evolution of the training procedure is in the correct direction (Figure 3.10) while it is apparent from the classification error of the training and test sets that overfitting occurs. Accuracy improvement may be obtained through a variety of approaches, such as proposing a better optimisation method, tuning the hyperparameters of the network [18], applying early stopping, using more data, and considering other deep learning architectures. Exploration of these potential improvements requires further investigation.



(a)



(b)

Figure 3.7: (a-b) Filters (or weights) the first layer learnt after the training procedure of the stacked autoencoder. Both images are the same but with a different color.

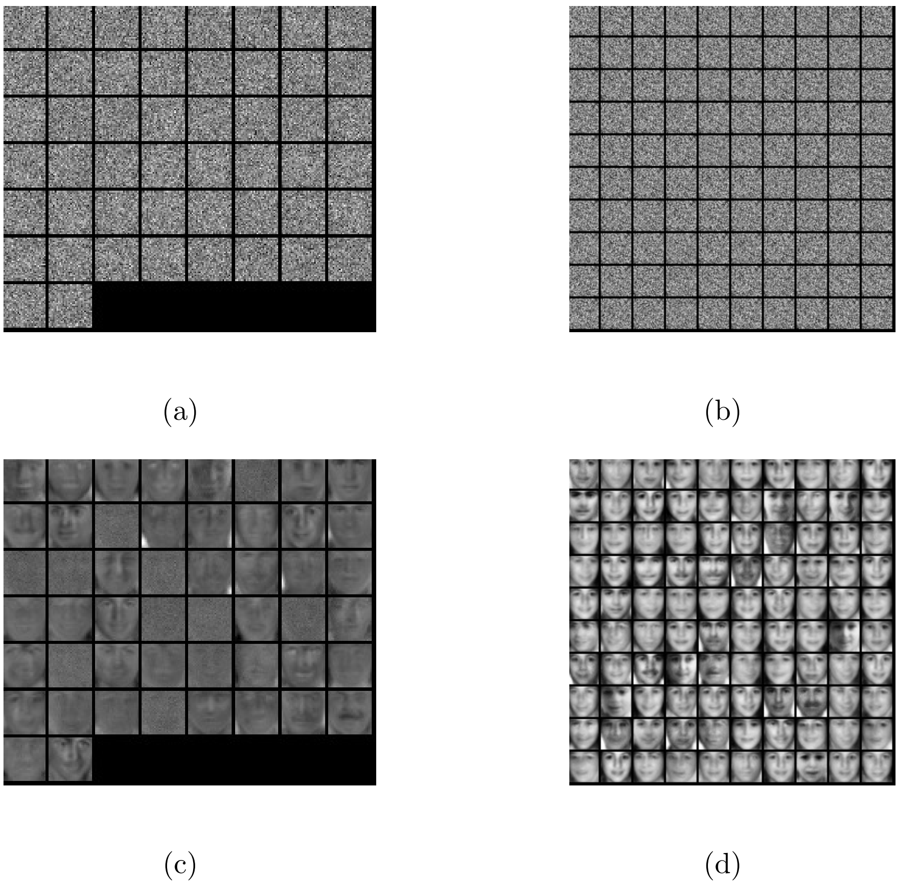
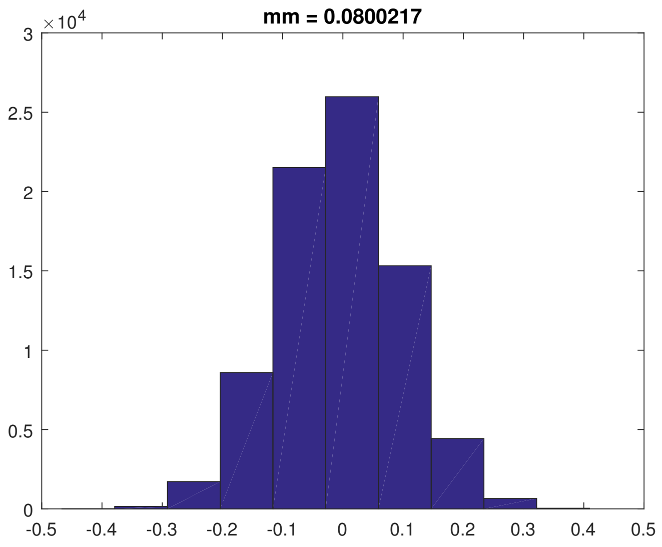
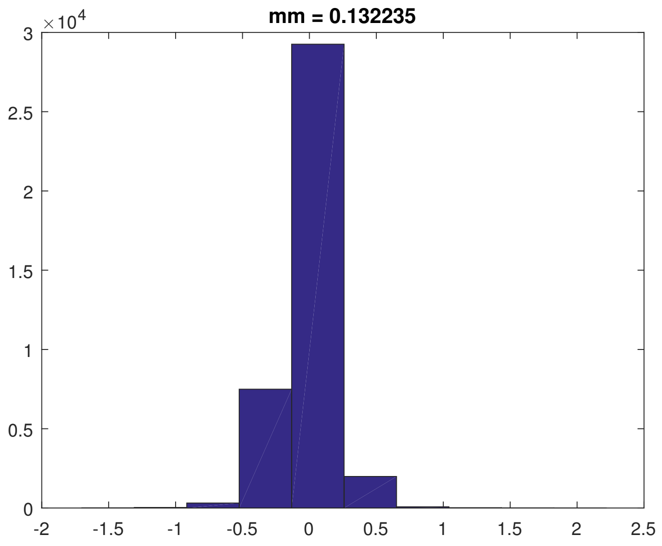


Figure 3.8: Initial (a-b) and final (c-d) of the filters learnt, respectively of the deep belief network.

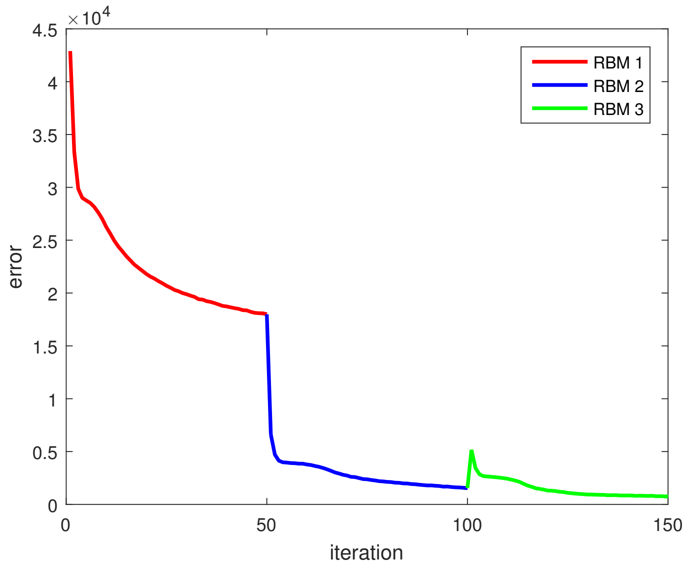


(a)

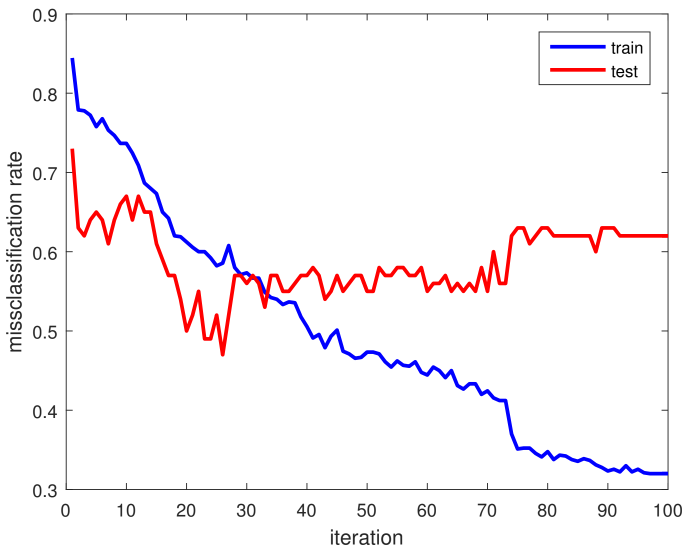


(b)

FIGURE 3.9: Histogram of filters at the beginning (a) and (b) end of the training. The mean absolute magnitude of the values is shown above each plot [106] .



(a)



(b)

FIGURE 3.10: (a) The error over the number of iterations and (b) the classification error for the training and test sets demonstrating that the model is overfitted.

3.3 Replacing the Metropolis-Hastings with importance sampling and resampling on the Radial Basis Function network

The Radial Basis Function (RBF) network is a neural network where the output of the network is a linear combination of the activation functions of the inputs and neuron parameters. The activation functions are radial basis functions. In this section, the model description of the RBF is discussed following the notation described in [10], which uses a hybrid MCMC to train the network with the core algorithms of the MH and Gibbs sampling. A new methodology is proposed where the MH is replaced with importance sampling and resampling and applied to the training procedure.

3.3.1 Model description

Consider the following model (provided in [10]):

$$
\begin{aligned}
&M_0 : y_t = b + \beta' x_t + n_t && k = 0 \\
&M_k : y_t = \sum_{j=1}^{k} a_j \phi(\|x_t - \mu_j\|) + b_t + \beta' x_t + n_t && k \geq 1
\end{aligned}
$$

or in the general form

$$y_t = f(x_t) + n_t$$

where $\|\cdot\|$ denotes the Euclidean distance metric, $\mu_j \in \mathbb{R}^d$ denotes the j^{th} Radial Basis Function (RBF) center for a model with k RBFs, $a_j \in \mathbb{R}^c$ the j^{th} RBF amplitude, $b \in \mathbb{R}^d$, $\beta \in R^d \times R^c$ the linear regression parameters and $n_t \in \mathbb{R}^c$ is a zero mean white Gaussian noise. The Gaussian noise is statistical noise having a probability density function equal to the Gaussian distribution. $x_t \in \mathbb{R}^d$ and $y_t \in \mathbb{R}^d$ represent the group of input and output variables, respectively. The variable $t = 1, 2, 3, \ldots$ corresponds to an index over the data.

The learning problem involves the approximation of the function $f(x_t)$ and estimation of the noise process given a set of input-output observations. The model can be expressed in the vector-matrix form

$$y = D(\mu_{1:k,1:d}, x_{1:N,1:d})\alpha_{1:1+d+k,1:c} + n_t \tag{3.13}$$

where D represents the regression matrix, α the linear regression parameters vector and $n_t \sim \mathcal{N}(0_{c\times 1}, \mathrm{diag}(\sigma_1^2, ..., \sigma_c^2))$ the zero mean white Gaussian noise.

$$\begin{bmatrix} y_{1,1}\dots y_{1,c} \\ \vdots \\ y_{N,1}\dots y_{N,c} \end{bmatrix} = \begin{bmatrix} 1 & x_{1,1}\dots x_{1,d} & \phi(x_1,\mu_1)\dots\phi(x_1,\mu_k) \\ \vdots & \vdots & \vdots \\ 1 & x_{N,1}\dots x_{N,d} & \phi(x_N,\mu_1)\dots\phi(x_N,\mu_k) \end{bmatrix}$$

$$\times \begin{bmatrix} b_1\dots b_c \\ \beta_{1,1}\dots\beta_{1,c} \\ \vdots \\ \beta_d,1\dots\beta_d,c \\ a_{1,1}\dots a_{1,c} \\ \vdots \\ a_{k,1}\dots a_{k,c} \end{bmatrix} + n_t \tag{3.14}$$

The following list includes different types of radial basis functions:

- Linear: $\phi(\rho) = \rho$
- Cubic: $\phi(\rho) = \rho^3$
- Thin plate spline: $\phi(\rho) = \rho^2 \ln(\rho)$
- Multiquadric: $\phi(\rho) = \sqrt{(\rho^2 + \lambda^2)}$
- Inverse quadratic: $\phi(\rho) = \frac{1}{(\rho^2+\lambda^2)}$
- Inverse Multiquadric: $\phi(\rho) = \frac{1}{\sqrt{(\rho^2+\lambda^2)}}$
- Gaussian: $\phi(\rho) = \exp(-\lambda\rho^2)$

where the λ is a user defined parameter. The minimum number of columns of the regression matrix, D, is 3, corresponding to a single RBF center (each column of this matrix apart from the first two describe the connections of the inputs with the RBF centers).

3.3.2 Bayesian aims using the hybrid MCMC

Bayesian inference is considered for the model, and the posterior distribution is computed using the expression (from Equation 3.3 in [10]):

$$p(k,\mu_{1:k},\Lambda,\delta|x,y) \propto \left[\prod_{i=1}^{c}(1+\delta_i^2)^{-m/2}(\frac{\gamma_0 + y'_{1:N,i}P_{i,k}y_{1:N}}{2})^{(-\frac{N+v_0}{2})}\right]$$
$$\times \left[\frac{\mathbb{I}_\Omega(k,\mu_k)}{\mathbb{J}^k}\right]\left[\frac{\Lambda^k/k!}{\sum_{j=0}^{k_{max}}\Lambda^j/j!}\right]\left[\prod_{i=1}^{c}(\delta_i^2)^{-(\alpha_{\delta^2}+1)}\exp(-\frac{\beta_{\delta^2}}{\delta_i^2})\right]$$
$$\times \left[(\Lambda)^{(\epsilon_1-1/2)}\exp(-\epsilon_2\Lambda)\right] \tag{3.15}$$

where k denotes the total number of RBFs, the $\mu_{1:k}$ the RBF centre, Λ, δ are hyperparameters, x, y are the inputs and outputs, respectively, and

$$M_{i,k}^{-1} = D^{\intercal}(\mu_{1:k}, x)D(\mu_{1:k}, x) + \Sigma_i^{-1} \tag{3.16}$$

$$h_{i,k} = M_{i,k}D^{\intercal}(\mu_{1:k}, x)y_{1:N,i} \tag{3.17}$$

$$P_{i,k} = I_N - D(\mu_{1:k}, x)M_{i,k}D^{\intercal}(\mu_{1:k}, x) \tag{3.18}$$

The likelihood is

$$p(y|k, \theta, \psi, x) = \prod_{i=1}^{c} p(y_{1:N,i}|k, \alpha_{1:m,i}, \mu_{1:k}, \sigma_i^2, x) \tag{3.19}$$

$$= \prod_{i=1}^{c} (2\pi\sigma_i^2)^{-N/2} \exp\Big(- \frac{1}{2\sigma_i^2}(y_{1:N,i} - D(\mu_{1:k}, x)\alpha_{1:m,i})^{\intercal} \tag{3.20}$$

$$\times\ (y_{1:N,i} - D(\mu_{1:k,x})\alpha_{1:m,i})\Big) \tag{3.21}$$

and the prior

$$p(k, a_{1:m}, \mu_{1:k}|\sigma^2, \Lambda, \delta^2) = p(a_{1:m}|k, \mu_{1:k}, \sigma^2, \delta^2)p(\mu_{1:k}|k)p(k|\Lambda) \tag{3.22}$$

$$= \left[\prod_{i=1}^{c} |2\pi\sigma_i^2\Sigma_i|^{-1/2} exp(-\frac{1}{2\sigma_i^2}\alpha'_{1:m,i}\Sigma_i^{-1}\alpha_{1:m,i})\right] \tag{3.23}$$

$$\times \left[\frac{\mathbb{I}_\Omega(k, \mu_{1:k})}{\mathbb{J}^k}\right]\left[\frac{\Lambda^k/k!}{\sum_{j=0}^{k_{max}} \Lambda^j/j!}\right] \tag{3.24}$$

This approach aims to estimate the posterior distribution for performing statistical inference using the predictive density

$$p(y_{N+1}|x_{1:N+1}, y_{1:N}) = \int_{\Theta\times\Psi} p(y_{N+1}|k, \theta, \psi, x_{N+1})p(k, \theta, \psi|x_{1:N}, y_{1:N})dkd\theta d\psi \tag{3.25}$$

Estimations are performed using

$$\mathbb{E}(y_{N+1}|x_{1:N+1}, y_{1:N}) = \int_{\Theta\times\Psi} D(\mu_{1:k}, x_{N+1})\alpha_{1:m}p(k, \theta, \psi|x_{1:N}, y_{1:N})dkd\theta d\psi \tag{3.26}$$

These quantities cannot be computed analytically as they require the evaluation of high dimensional integrals of nonlinear functions in the parameters.

In the equations, the k defines the number of RBFs and in our experiments it is considered a fixed value (identically with Method 4.1 in [10]). The Λ, δ are the hyperparameters.

$$\sigma_i^2|(k, \mu_{1:k}, \delta^2, x, y) \sim \mathbb{IG}\left(\frac{\upsilon_0 + N}{2}, \frac{\gamma_0 + y'_{1:N,i} P_{i,k} y_{1:N,i}}{2}\right) \tag{3.27}$$

$$\alpha_{1:m,i}|(k, \mu_{1:k}, \sigma^2, \delta^2, x, y) \sim \mathbb{N}(h_{i,k}, \sigma_i^2, M_{i,k}) \tag{3.28}$$

$$\begin{aligned}\delta_i^2|(k, \alpha_{1:m}, \mu_{1:k}, \sigma_i^2, x, y) \sim \mathbb{IG}\Big(a_{\delta^2} + \frac{m}{2},\\ \beta_{\delta^2} + \frac{1}{2\sigma_i^2} a'_{1:m,i} D'(\mu_{1:k}, x) D(\mu_{1:k}, x) \alpha_{1:m,i}\Big)\end{aligned} \tag{3.29}$$

$$q(\Lambda^*) \propto \Lambda^{*(1/2+\epsilon_1+k)} exp(-(1+\epsilon_2)\Lambda^*) \tag{3.30}$$

where $\mathbb{IG}(a, b)$ is the inverse Gamma distribution with mean value a and shape parameter b. The $\Theta \times \Psi$ define the overall parameter space which is described as a finite union of subspaces $\Theta \times \Psi = (\bigcup_{k=0}^{k_{max}} \{k\} \times \Theta) \times \Psi$, where $\Theta_0 \triangleq (\mathbb{R}^{d+1})^c \times (\mathbb{R}^+)^c$ and $\Theta_k = (\mathbb{R}^{d+1+k})^c \times (\mathbb{R}^+)^c \times \Omega_k$, for $k = 1, 2, \ldots, k_{max}$, $\alpha \in (\mathbb{R}^{d+1+k})^c$, $\sigma \in (\mathbb{R}^+)^c$ and $\mu \in \Omega_k$. The hyperparameters include the space $\Psi \triangleq (\mathbb{R}^+)^{c+1}$, with elements $\psi = \{\Lambda, \delta^2\}$.

Algorithm 8 describes the Hybrid MCMC (Method 4.1 in [10]). In every iteration of the Hybrid MCMC, the centres are sequentially updated using the MH algorithm with an acceptance rate

$$a_r = \frac{P}{P_p} \tag{3.31}$$

where

$$P = \left(\prod_{i=1}^{c} (\gamma_0 + y_{1:N,i}^{\intercal} P_{i,k} y_{1:N,i})\right)^{\frac{N+\upsilon_0}{2}} \tag{3.32}$$

The denominator p refers to the proposal density and depends on the condition (see steps 5 and 6 in Algorithm 8). The first proposal (step 5) is a random walk, which is a Gaussian distribution with a mean value of the centre of the RBF and a user-defined covariance, c_u [10]

$$q_1(\mu_{j,1:d}^*|\mu_{j,1:d}^{(i)}) = \mathcal{N}(\mu_{j,1:d}^{(i)}, c_u) \tag{3.33}$$

Otherwise, the proposal (step 6) is selected with a uniform distribution in the interval $(\min(x) - w, \max(x) + w)$, where w is a user-defined parameter [10]

$$q_2(\mu^*_{j,1:d}|\mu^{(i)}_{j,1:d}) = (\min(x) - w, \max(x) + w) \tag{3.34}$$

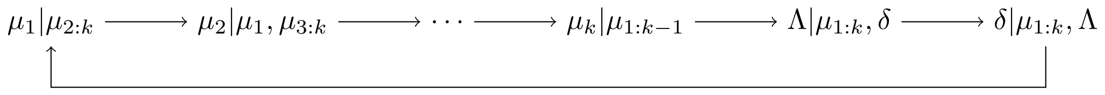
Algorithm 8 Hybrid MCMC on RBFs

```
1: Initialisation. Fix the value of $k$ (number of RBF centres)and set $(\theta^{(0)}, \psi^{(0)})$
2: for $i = 1 : T$ do
3:     for $j = 1 : k_{max}$ do
4:         Sample $u \sim U_{[0,1]}$
5:         If $u > 0.5$, perform an MH step admitting $p(\mu_{j,1:d}|x, y, \mu^{(i)}_{-j,1:d})$ as invariant
           distribution and $q_1(\mu^*_{j,1:d}|\mu^{(i)}_{j,1:d})$ as proposal distribution
6:         Else perform an MH step using $p(\mu_{j,1:d}|x, y, \mu^{(i)}_{-j,1:d})$ as invariant distribution
           and $q_2(\mu^*_{j,1:d}|\mu^{(i)}_{j,1:d})$ as proposal distribution.
7:     end for
8:     Sample the nuisance parameters $(\alpha^{(i)}_{1:m}, \sigma^{2(i)})$ using Equations 3.27 and 3.28.
9:     Sample the hyperameters $(\Lambda^{(i)}, \delta^{2(i)})$ using Equations 3.29 and 3.30
10: end for
```

3.3.3 Proposed method on the RBF

As discussed the initial method proposed in [10] assumes a fixed number of RBF centers (Algorithm 8). There are several potential approaches for using SMC samplers with this problem, and the key consideration is to replace the MH steps with an SMC sampler.

The i^{th} iteration of the Hybrid MCMC method (Algorithm 8) consists of a sequence of operations (Figure 3.11) where each RBF centre is sequentially updated based on the current RBF centres with the parameters and hyperparameters.



$$\begin{tikzcd} \mu_1|\mu_{2:k} \arrow[r] & \mu_2|\mu_1, \mu_{3:k} \arrow[r] & \cdots \arrow[r] & \mu_k|\mu_{1:k-1} \arrow[r] & \Lambda|\mu_{1:k}, \delta \arrow[r] & \delta|\mu_{1:k}, \Lambda \arrow[lllll, bend left] \end{tikzcd}$$

FIGURE 3.11: The diagram describes the sequence of operations for the *ith* Hybrid MCMC iteration (Algorithm 8).

The identical chain of operations is computed in the RBF using importance sampling and resampling. Each particle represents the model and follows the operations in Figure 3.11. In the Hybrid MCMC, the centres of the RBFs are updated using the MH method, and there is a single regression matrix. In the proposed method, each RBF centre is instead updated using a single SMC sampler step, and every particle uses its regression matrix. Denoting N_p as the total number of particles, each defines a separate model. There are $D_n \in \mathbb{R}^{N_p \times 1}$ total regression matrices, where $n = \{1, 2, \ldots, N_p\}$. Equivalently, each particle has parameters, $\alpha^{(n)}$, $\sigma^{2^{(n)}}$ and hyperparameters $\Lambda^{(n)}$, $\delta^{2^{(n)}}$, and is associated

Algorithm 9 ISR on RBFs

```
1: Initialisation. Fix the value of $k$ (number of RBF centres) and set $(\theta^{(0)^{(1:N)}}, \psi^{(0)^{(1:N)}})$
2: for $i = 2 : T$ do
3:     for $n = 1 : N$ do
4:         for $j = 1 : k_{max}$ do
5:             Sample $u \sim U_{[0,1]}$
6:             If $u > 0.5$, update the RBF centers using $q_1(\mu^{*(i,n)}_{j,1:d} | \mu^{(i,n)}_{j,1:d})$ as proposal
               distribution
7:             Else update the RBF centers $q_2(\mu^{*}_{j,1:d}{}^{*(i,n)} | \mu^{(i,n)}_{j,1:d})$ as proposal distribution.
8:         end for
9:     end for
10:    Compute importance weights using Equation 3.35
11:    Compute the effective sample size and resample if needed (see Section 2.4.1)
12:    for $n = 1 : N$ do
13:        Sample the nuisance parameters $(\alpha^{(i,n)}_{1:m}, \sigma^{2^{(i,n)}})$ using Equations 3.27 and 3.28.
14:        Sample the hyperameters $(\Lambda^{(i,n)}, \delta^{2^{(i,n)}})$ using Equations 3.29 and 3.30
15:    end for
16: end for
```

with a weight. In the weight calculation it is assumed that the backward kernel is equal with the forward kernel and, thus, eliminated from the calculation. The pseudocode for this method is available in Algorithm 9.

$$w_k^{(n)} = \frac{P^{(n)}}{P_p^{(n)}} \tag{3.35}$$

After the weights calculation using Equation 3.35 the particles are resampled so that negligible weights are eliminated and replicated with more important weights. Essentially, the new method corresponds to replacing the MH with the initial step of the SMC sampler consisting of importance sampling and resampling steps (ISR).

3.3.4 Signal detection experiments

In this problem, data are generated from a univariate function using 50 covariance points on $[-2, 2]$ [10]:

$$y = x + 2\exp(-16x^2) + 2\exp(-16(x - 0.7)^2) + n$$

where $n \sim \mathbb{N}(0, \sigma^2)$. The data are rescaled to make the input lie within the interval $[0, 1]$. The radial basis functions are Gaussian with the same variance as the Gaussian signal noise. Figure 3.12 shows the fits obtained for the training and test sets from both methods, and Figure 3.13 shows the accuracy benefits of increasing the number of particles. The results are provided by varying the variance of the noise σ^2 in Table 3.1

with a comparison of the proposed method with the hybrid MCMC algorithm from [10]. In Tables 3.2 and 3.3, the new method is compared with different choices of the number of particles and RBF centres, respectively.

In the comparison of the two methods, it can be seen that the hybrid MCMC is better than the proposed algorithm due to the different evaluation of the importance weights of the proposed method in contrast the basic SMC . The proposed methodology does not consider the weights of the previous iteration, but only the identical used as acceptance probability in the Hybrid methodology (i.e. Algorithm 8) in the Metropolis-Hastings. Essentially, this is correlated with Chapters 2 and 7, where comparison of the basic SMC sampler with other methodologies demonstrates worse accuracy than the MH algorithm, while better versions of the same algorithm outperform the SMC sampler and competitor methods. Another observation is that even on this toy example when the number of the particles increases, the sequential computational time of the proposed method is significantly worse compared to that in [10]. Specifically, on a non-optimised, sequential, code the runtime is roughly three times slower than the original method. The time difference depends on the number of particles and the total number of RBF centres. This issue is examined in detail in Chapter 4, where a new fully distributed algorithm for the resampling is proposed.

TABLE 3.1: Root Mean Squared Error (RMSE) for Different Noise Values ($N = 100$ and $k = 2$ RBF centers

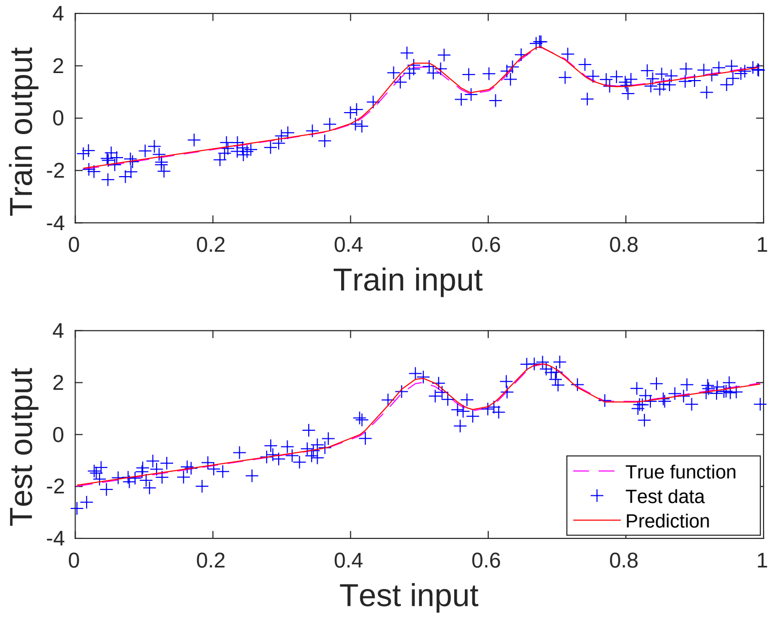
σ^2	Hybrid Method [10] (RMSE)	Proposed Method (RMSE)
0.01	0.025	0.18
0.1	0.052	0.22
1	0.25	0.39

TABLE 3.2: Root Mean Squared Error (RMSE) for Different Number of Particles ($\sigma^2 = 0.1$ and $k = 2$)

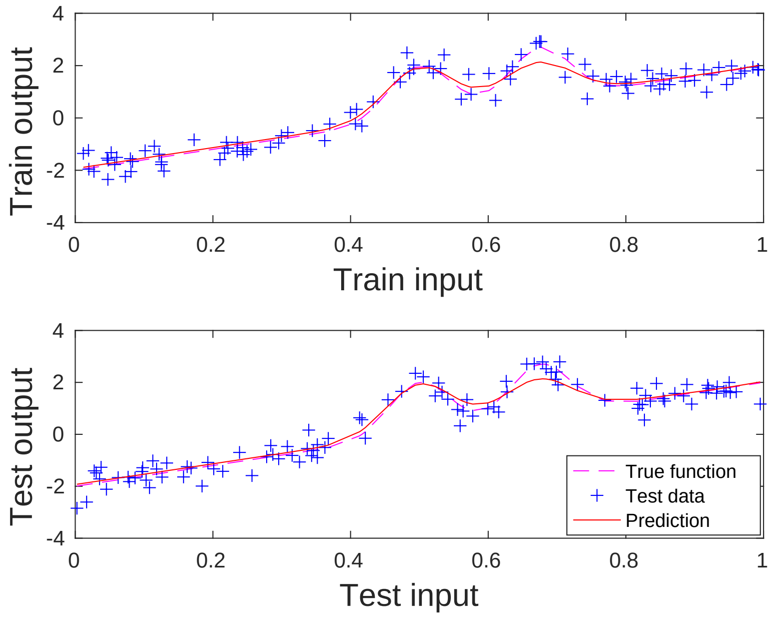
Number of Particles (N)	Proposed Method (RMSE)
10	0.33
100	0.22
1000	0.12

TABLE 3.3: Root Mean Squared Error (RMSE) for Different Number of RBF Centers ($N = 100$ and $\sigma^2 = 0.1$)

RBF Centers (k)	Proposed Method (RMSE)
1	0.45
2	0.22
4	0.13

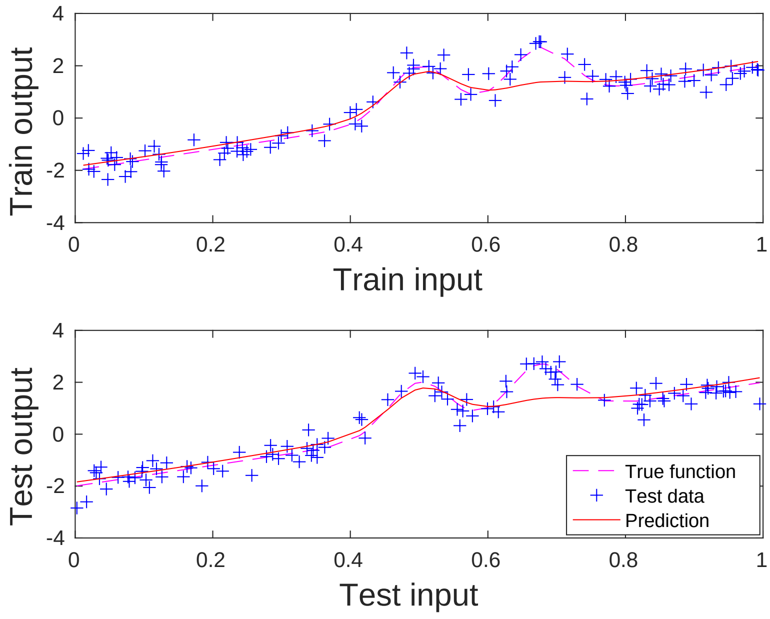


(a)

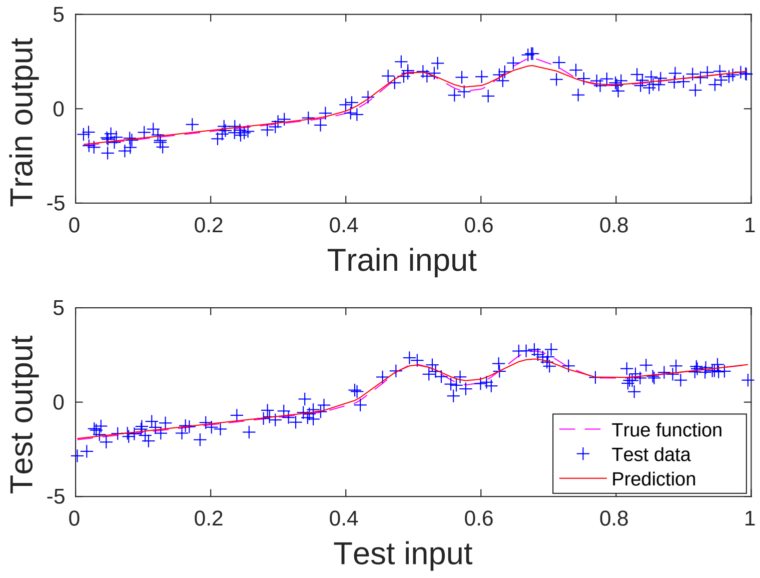


(b)

FIGURE 3.12: Example of the performance of (a) the hybrid MCMC algorithm and (b) the proposed method using the same input data.



(a)



(b)

FIGURE 3.13: Example of the performance using (a) 10 particles and (b) 1000 particles for the same input data.

3.4 Conclusions

In this chapter, we overviewed two traditional deep learning algorithms including a discussion of examples of training procedures based on stochastic and deterministic approaches with an emphasis on stochastic networks. In the last section, our effort focused on training a neural network using an SMC method. The proposed method revealed issues related to its accuracy as well as the computation time required to execute a benchmark experiment. In the following chapters, the research is concentrated on strategies for improving the performance and accuracy of SMC methods.

Chapter 4

Parallel sequential Monte Carlo methods

4.1 Introduction

In this chapter, a novel resampling algorithm is proposed and applied on the Sequential Importance Resampling (SIR) particle filter, a Sequential Monte Carlo (SMC) method, using two programming models, MapReduce and Message Passing Interface (MPI).

SMC methods have the appealing property that, as the number of samples increases, the ability of the samples to represent the probability density function (pdf) increases and the accuracy of estimates derived from the particles improves with an upper-bound on the variance of the estimate scaling as $O\left(\frac{1}{N}\right)$ [59]. It is, therefore, reasonable to use as many particles as possible. The resampling component is critical for SMC methods and non-trivial to parallelise. The aim is to make the resampling process more amenable to a distributed implementation. A textbook implementation of this algorithm solves the problem in $O(N)$ operations but is not suitable for a fully balanced, multi-core implementation. The proposed method improves the time complexity of previous work from $O((\log N)^3)$ to $O((\log N)^2)$.

In Section 4.2, the parallel resampling algorithms is reviewed. Section 4.3 includes the proposed novel parallel resampling implemented in MapReduce, and Section 4.3 is based on [97]. The results indicate that Apache Spark provides significantly better run-time over the Apache Hadoop implementation as it enables the random access memory (RAM) to store the data. However, it does not yet outperform the linear time solution on MapReduce, even with a considerable degree of parallelism. The algorithm is reformulated for distributed memory setup using MPI and is available in Appendix A (conference publication [101]). The results indicate that MPI is more suitable than MapReduce for this method.

4.2 Review on parallel resampling

The resampling algorithm is the solution to the degeneracy phenomenon in SMC methods (Particle filters and SMC samplers) [12] [29], which outperform the Kalman filter (optimal for linear solutions) and extended versions of the Kalman filter in nonlinear scenarios [66] [54]. The resampling is the bottleneck to the parallel execution with $O(N)$[1] sequential time complexity. Most of the publications consider parallel versions of the resampling algorithm as part of the overall particle filter, although some exceptions exist that focus solely on the resampling.

The parallel resampling is classified with methods intended to improve the time complexity of the traditional algorithms (i.e., multinomial, stratified, systematic, and residual resampling) or to propose alternative methodologies more easily parallelised, and that remove components such as the prefix sum (e.g., MH and Rejection resampling). Another category is based on the distribution of the computational workload in centralised, partially centralised, and distributed strategies. An example of partially centralised computation is the resampling with proportional allocation (RPA), and an example of distributed computation is the resampling with nonproportional allocation (RNA). A special case of distributed computation is load balancing or deterministic methods. In [56], a detailed review classifies the resampling algorithms as sequential or parallel/distributed, and based on parallel platforms used (e.g., VLSI, GPU, and FPGA).

In Table 4.1, multiple publications are grouped according to the resampling method as biased or unbiased. In biased methods, the new population of particles approximates to the old population, and, as a result, they introduce larger error in the final estimation. The performance and accuracy comparison of the traditional methods is analysed in [47]. The results indicate that by increasing the number of particles, all methods have identical accuracy, while in the sequential case, the systematic resampling is fastest. In the multinomial resampling, also known as simple random resampling [60], the particles are selected randomly and uniformly. The stratified resampling divides the particles into even subpopulations, named strata, and for each stratum, an offset is selected. In the systematic resampling for all strata, a unique offset is selected. The multinomial, stratified, and systematic resampling methods require the computation of the prefix sum of the normalised weights. For the parallel traditional resampling methods, the reported time complexities range from $O(\log N)$ to $O(N)$[2] solutions. The parallel resampling is applied with a range of platforms including hardware and software implementations. For the MH and Rejection methods, the reported time complexity ranges from $O(B)$ to $O(N)$ (B denotes the number of iterations) and $O(\log N)$ to $O(N)$, respectively. Further details provided by the selected publications include the maximum number of particles in logarithmic form, the maximum state dimensionality of the application, the reported maximum speedup including the efficiency, and the baseline for the reported

[1]N indicates the number of particles.

[2]sequential execution of the algorithm.

maximum speed up. Three trends are considered for the baseline, including executing the method on a single core, single node or processing unit, and different platform. Other information is related to the applied parallel approach for the cumulative summation [20] and the redistributed components classified as Fully Distributed and Deterministic. The time complexity of the fully distributed cumulative sum is $O(\log N)$. A deterministic redistribute method implies balanced computational load with deterministic runtime (e.g., [97] [49]), and a deterministic runtime is essential for real-time applications.

In [41], stratified resampling is used on the SIR particle filter on a GPU implementation. The time complexity of the resampling is $O(\log N)$ on a hardware implementation (rasterizer) for the particle redistribution. In [63], the redistribute method in the systematic resampling implies the use of binary search for each particle in the (monotonically increasing) prefix sum vector. The computational attention is focused on the sampling stage, which appears to be the most expensive component of the particle filter. In contrast, the implementation in [36] of a fully distributed version of the systematic resampling using a GPU implementation shows that the resampling remains the bottleneck as well as in the parallel version. A potential explanation is that the sampling stage considered in the benchmark in [36] is computationally trivial where the prefix sum is parallelised using the fully distributed approach discussed in [20]. The data dependency of the sequential redistribute algorithm is removed by introducing two variables, a left and right boundary determined from the prefix sum computation of the normalised weights and a uniform random number [36]. A load balancing systematic resampling is proposed in [49] and compared to traditional systematic resampling. The parallelisation of the prefix sum is based on non-fully distributed approach (also available in [20]) with a separation of the weights distributions into blocks, local sum computation, global prefix sum, and local per element sum according to the global prefix sum. In [49] each thread block, B, uses L threads and in total there are $N = LB$ particles where a thread is assigned to each particle. In the traditional redistribute, each particle has a replication index (or there is a replication index array for all particles), and each thread generates copies according to the assigned replication index. This is inefficient because in the worst case a single thread might have a replication index of N leading to linear time complexity for the particles generation. The proposed methodology reduces the complexity by creating a minimal replication index array which can recover the original array as in the traditional algorithm. The minimal replication array forms groups of the same indices for particles requiring multiple replications. The recovery is achieved using a balanced binary tree with time complexity $O(\log L)$ and removes the imbalance among the threads during the generation of the new population of particles. In the worst case, the algorithm' time complexity is improved from $O(N)$ to $O(\frac{N}{L})$. The results indicate that the runtime fluctuations of the proposed method are significantly smaller than the traditional approach. In [14], MapReduce is applied with the particle filter where the sampling and weight normalisation steps are parallelised, while the resampling is executed sequentially. In [101, 97, 67], the resampling is executed in a fully distributed fashion. The key difference between [67] and [101, 97] is that the time complexity is improved from $O((\log N)^3)$ to $O((\log N)^2)$ after including a method to eliminate the need to sort the replicated particles in every step (or level the balanced binary tree) in the redistribute algorithm. In [101], the MPI/HPC-based solution is shown to be better than MapReduce [97].

The RPA/RNA methods with a centralised resampling are proposed in [21]. In the centralised resampling, a single processing unit (called a central unit) executes the resampling sequentially. In this category of methodologies, resampling with proportional allocation (RPA) is the first attempt to parallelise the resampling method. The RPA is grouped with the traditional algorithms. Comparing the same input to any traditional resampling and the equivalent RPA, both methods yield the same output. In the RPA, the resampling is partially executed in the central unit, which decides the replications (number of copies) for each processing unit. Then, each processing unit is assigned the task to create the replications for each particle. After the execution of the resampling, single or multiple processing elements (PEs) can result in zero particle replications. After the resampling, the particles are then exchanged among the PEs for an equivalent distribution of particles. The worst communication scenario is a single PE for replicating all particles. In the RNA method, each PE executes resampling locally. Comparison of this approach with any traditional method will not yield the same output. Groups of PEs with a predefined number of particles are formed and perform within each group sampling and resampling. After the local (i.e., within each group) resampling step, particles are exchanged so that all groups contain equally distributed weights. Multiple particle routing strategies are proposed. The RPA is computationally more expensive compared to the RNA as it is partially executed in the central unit. In the RNA, the particle routing is deterministic and non-deterministic in the RPA. The RPA discussed in [21] is applied in [110] with MPI. The methodologies proposed in [15] and [95] follow a similar approach with the RPA and RNA methodologies.

Biased resampling methods (e.g., [91], [70] and [34]) provide computational and communication complexity improvements, but with a cost in accuracy. In [73], the MH and Rejection methods are proposed as alternatives to the traditional methodologies, and a comparison highlights their numerical stability. The reported time complexity of the Rejection resampling is $\frac{logN}{r}$, where $r = \min_i(\frac{\frac{1}{N}\sum_i w^i}{w^i}) \cdot \max_i(\frac{w^i}{w_{\max}})$, with $i = 1, \ldots, N$ and $O(B)$ for the Metropolis resampling. If the variable B in the MH is selected appropriately, then it is faster than the Rejection resampling. A similar analysis on using the MH and Rejection resampling methods is conducted in [62] on FPGAs. Practically, the time complexity of both methods is determined from the number of trials needed to draw the new particle, which can be a function of the total number of particles (i.e. $O(N)$ time complexity). In [91], it is demonstrated on a widely applicable benchmark [12] that for a small value of B, such that $B \ll N$, the distance, or error, is negligible for the final estimation with the systematic resampling method.

Several publications on parallelising particle filters ignore the importance of the resampling algorithm focusing, for instance, on parallelisation of the SIS particle filter (e.g., [61]).

TABLE 4.1: Publications grouped according to the applied parallel resampling methodology with its reported time complexity (TC).

Paper	Year	Platform	Method	Biased	TC
[101]	2017	MPI	Systematic	N	$O((logN)^2)$
[97]	2017	MapReduce	Systematic	N	$O((logN)^2)$
[63]	2015	GPU	Systematic	N	$O(logN)$
[105]	2015	GPU	Systematic	N	$O(\frac{N}{P})$
[62]	2014	FPGA	Systematic	N	$O(\frac{N}{M})$
[49]	2013	GPU	Systematic	N	$O(\frac{N}{P})$
[36]	2012	GPU	Systematic	N	$O(logN)$
[14]	2012	MapReduce	Systematic	N	$O(N)$
[67]	2006	Graphics (CG)	Systematic	N	$O((logN)^3)$
[110]	2016	MPI	RPA	N	$O(\frac{N}{P})$
[95]	2012	Simulation	RNA/RPA	N	$O(\frac{N}{P})$
[52]	2011	MPI	RPA/RNA	N	$O(\frac{N}{P})$
[92]	2006	VLSI	RPA/RNA	N	$O(\frac{M}{K})$
[21]	2005	FPGA	RPA/RNA	N	$O(\frac{N}{P})$
[15]	2003	Simulation	RPA/RNA	N	$O(\frac{N}{P})$
[41]	2010	GPU	Stratified	N	$O(logN)$
[96]	2016	FPGA	Metropolis	Y	$O(B)$
[73]	2016	GPU	Metropolis	Y	$O(B)$
[91]	2015	FPGA	Metropolis	Y	$O(B)$
[62]	2014	FPGA	Metropolis	Y	$O(\frac{BN}{M})$
[70]	2010	FPGA	Metropolis	Y	$O(N)$
[73]	2016	GPU	Rejection	N	$O(\frac{logN}{r})$
[62]	2014	FPGA	Rejection	N	$O(\frac{SN}{M})$

4.3 MapReduce particle filtering with exact resampling and deterministic runtime

4.3.1 Introduction

This section describes an implementation of a particle filter using MapReduce and focuses on the resampling component, which would otherwise be a bottleneck to parallel execution. We devise a new implementation of this component that requires no approximations, has $O(N)$ spatial complexity, and deterministic $O\left((\log N)^2\right)$ time complexity. The results demonstrate the utility of this new component culminating in consideration of a particle filter with 2^{24} particles distributed across 512 processor cores. The key contributions include:

- An improved implementation of an exact deterministic resampling algorithm with better temporal complexity compared to the current state of the art [67]. More specifically, the proposed version of the parallel algorithm has the complexity of $\mathcal{O}((\log_2 N)^2)$ compared to the original complexity of $\mathcal{O}((\log_2 N)^3)$.

- Two MapReduce variants of the new algorithm that fit with the in-memory and out-of-core processing models, which are the processing models used by Hadoop and Spark, respectively.

- A detailed performance and scalability analysis of the new algorithm in comparison to the existing state of the art [67] and an implementation optimised for a single processor core. We deliberately select an application that stresses the resampling component of the particle filter so that our analysis considers the worst-case performance.

The remainder of this chapter is organised as follows. Section 4.3.2 provides a brief overview of Big data processing and the MapReduce programming model. Section 4.3.3.1 describes the fundamental building blocks used to construct the implementations of the particle filtering algorithm, including, in Section 4.3.3.1.9, the new component of the resampling algorithm. The MapReduce-based particle filtering implementation is outlined in Section 4.3.4 followed by an evaluation of our algorithms on two important MapReduce frameworks in Section 4.3.5.

4.3.2 Big data processing

This section focuses on the problem of using a large number of samples within a particle filter. Big data processing frameworks (e.g., Apache projects, such as Hadoop [2], Spark [5] and Storm [6][3]) are designed for handling large amounts of data and can therefore be applied in this context[4]. We highlight such frameworks in conjunction with parallel computational resources, such as clusters, to handle large volumes of data[5]. In this section, we also discuss the use of such Big data frameworks in general, and, in particular, one of the programming models that underpins such frameworks, the MapReduce programming model.

4.3.2.1 Big data frameworks

An attractive approach for scaling the problem with data is to use Big Data frameworks, which go beyond the issue of data volume and address much wider issues covering the augmented V's of data, specifically *volume, velocity, variety, value* and *veracity* [90]. Big Data framework-based solutions are process-centric as the programmer describes the algorithm in a way that enables the framework to understand (and attempt to exploit) the potential to distribute the data and processing[6]. The result of this delegation of

[3]Including the associated ever-growing ecosystem of tools (e.g., Mahout [3] and GraphX for Spark [94]).

[4]Conventional High Performance Computing (HPC) approaches use parallel computations to optimise processing time. We refer the reader to [13] for a good coverage of HPC-bound approaches for parallelising applications.

[5]We anticipate that the 'heat wall' (i.e., the inability to remove enough heat from transistors that switch ever faster) will mean that for chip manufacturers to meet the expectation set by Moore's law, they will soon (If not already) be doubling the number of cores (not transistors per square inch) used in each processor each year. In ten years' time, if this trend continues, we would have desktop computers with a thousand times as many cores as today. This trend motivates the authors to design implementation strategies for particle filters that are well suited to the multi-core processors which will, we believe, become increasingly prevalent over time.

[6]This contrasts HPC-based solutions, where the programmer aims to exploit intricate knowledge of the underlying architecture to ensure that data movement and processing are jointly optimised for the specific hardware.

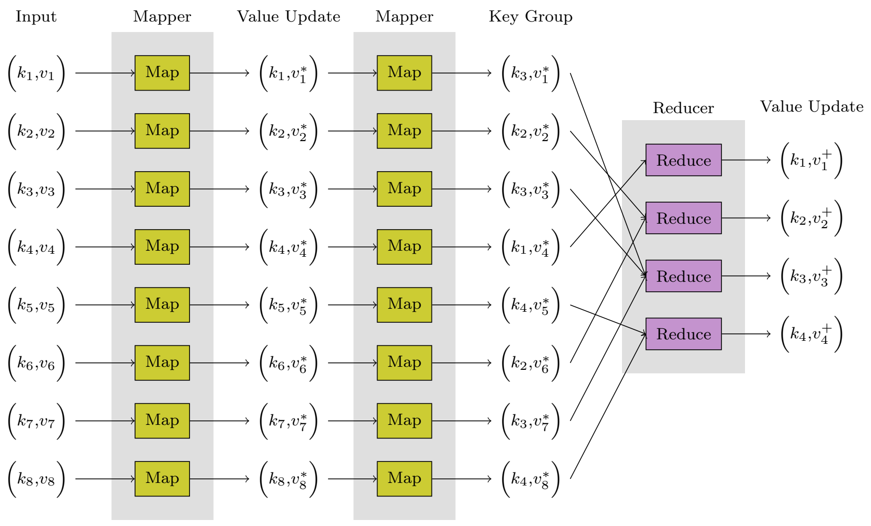


FIGURE 4.1: General MapReduce Processing Model.

the optimisation for speed to the framework is that, while many of today's Big Data frameworks can handle large volumes of data, none can match the runtime performance of conventional HPC systems [86]. There are a growing number of programming models used to describe algorithms within Big Data frameworks, including MapReduce [28], Stream Processing [42, 5, 6] and Query-based techniques [103, 4]. Here, we focus on MapReduce.

4.3.2.2 The MapReduce programming model

MapReduce is a popular programming model used in many big data processing frameworks (and even some HPC frameworks). The key idea of the MapReduce model is to enable the framework to distribute the processing of a large dataset by expressing algorithms in terms of *map* and *reduce* operations, via defining *mappers* and *reducers*. Mappers, when applied to each datum, output a list of *(key, value)* pairs. The framework then collates all the values associated with each key. Reducers are then applied to the list of values for each key to output a single value. Both the map and reduce operations are inherently parallel across all data and keys, respectively[7]. To exemplify this, consider a dataset where each datum is a sentence in a Big document (e.g., Wikipedia). The problem of counting the total number of occurrences of each word in the document corpus can be described as using the words as the key, a mapper that outputs a (non-zero) count of the number of times each word occurs in each sentence[8], and a reducer that calculates the sum of the counts. For each word, the reducer's output is the sum over all sentences of the counts per sentence. Another example is shown in Figure 4.1

[7]The exact number of mapper and reducer processes on a parallel resource (for instance, a multi-node cluster) varies depending on the configuration, but the important point is that the algorithm developer does not need to worry about how the processes are distributed when defining the algorithm. Of course, that does not mean that there is not utility in the developer describing algorithms using mappers and reducers that are well suited to the problem being tackled and to the configuration being used.

[8]Note that the output from each sentence would only be for the words that occur in that sentence, not every word that ever occurs in the corpus.

illustrating the ability to pass key-value pairs into a mapper and, thereby, use the output of one mapper as the input into a second mapper.

Two key frameworks that support MapReduce, albeit in slightly different ways, are Hadoop and Spark, which are considered in the following.

4.3.2.2.1 Hadoop

MapReduce and the Hadoop Distributed File System (HDFS) are the two fundamental components of Hadoop. HDFS enables multiple computers' disks to be accessed in much the same way as if it was a single (Big) disk. In Hadoop, the mapper and reducer generate files stored in HDFS, such that Hadoop implements data movement entirely via the file system.

4.3.2.2.2 Spark

The Spark framework operates using a different principle than that of Hadoop. First, at the Application Programming Interface (API) level, Spark provides a distributed data structure known as a Resilient Distributed Dataset (RDD) [108]. MapReduce is then just one of a large number of *transformations* that (via a rich set of APIs) can be applied to RDDs. It is also important to realise that evaluations in Spark are *lazily* executed. This means, unlike conventional processing engines (e.g., Hadoop), executions never actually happen when transformations are defined. Instead, transformations are used to compose a data-flow graph and execution happens when forced through *actions* (i.e., when necessary). This delayed evaluation enables the Spark framework to optimise (and plan) the execution[9]. The result is often significant improvements in runtime performance. Another important property of RDDs is they can reside in memory, disk or both. Indeed, although Spark can make use of HDFS, the data movements in Spark are primarily via memory resulting in significant improvements in runtime performance relative to Hadoop.

4.3.3 Parallel particle filtering

The bulk of the operations comprising the particle filter (as described in Algorithm 6) are readily parallelised. However, it is resampling (the redistribution process, in particular) that complicates the parallel implementation of particle filters.

Complications primarily arise because if each of the multiple processors is considering subsets of the particles, then the data transfers that the redistribution process demands are data-dependent. It is, therefore, non-trivial to implement a particle filter in a way that the run-time is not data-dependent. A similar problem is encountered with sorting algorithms[10]. In the subsequent sections of this chapter, we describe how to implement

[9]This can make it hard for a programmer to debug algorithmic implementations, particularly if the programmer is unfamiliar with debugging software performing lazy evaluation.

[10]For instance, although Quicksort [46] can be parallelised, the load distributions across the processors is dependent on the pivots used and the run-time will therefore be data-dependent.

the components of the particle filter in a way that run-time is deterministic instead of data dependent.

4.3.3.1 Parallel instantiations of the algorithmic components of particle filtering

Before mapping the particle filter algorithm on to a MapReduce form, it is essential to understand how the operations used by a particle filter can be implemented in a fully distributed form. While a detailed discussion of these operations (and others) is found in [20], we discuss here each of the operations that constitute the algorithm described in Algorithm 6. Table 4.2 summarises these operations and associated complexities for the fundamental building blocks and some of the algorithmic components that can be built. Our focus is on implementations with a time complexity that is as fast as possible in terms of its dependence on N, the number of data. We discuss communication complexity for each algorithmic component by considering a simplified memory architecture where transferring a datum between two processors is a single data movement.

Table 4.2: Theoretical complexities (in terms of time, space and total data transfers per unit time) of various algorithmic components of the Particle Filter with N data and P processors.

Section	Algorithmic Component	Time	Space	Data Transfers
4.3.3.1.1	Element-wise operations	$\mathcal{O}(1)$	$\mathcal{O}(N)$	$\mathcal{O}(1)$
4.3.3.1.2	Rotation	$\mathcal{O}(1)$	$\mathcal{O}(N)$	$\mathcal{O}(1)$
4.3.3.1.3	Sum/Max/Min	$\mathcal{O}(\frac{N}{P}\log N)$	$\mathcal{O}(N)$	$\mathcal{O}(P)$
4.3.3.1.4	Cumulative Sum	$\mathcal{O}(\frac{N}{P}\log N)$	$\mathcal{O}(N)$	$\mathcal{O}(P)$
4.3.3.1.5	Normalising the Weights	$\mathcal{O}(\frac{N}{P}\log N)$	$\mathcal{O}(N)$	$\mathcal{O}(P)$
4.3.3.1.6	Minimum Variance Resampling	$\mathcal{O}(\frac{N}{P}\log N)$	$\mathcal{O}(N)$	$\mathcal{O}(P)$
4.3.3.1.7	(Bitonic) Sort	$\mathcal{O}(\frac{N}{P}(\log N)^2)$	$\mathcal{O}(N)$	$\mathcal{O}(P)$
4.3.3.1.8	Redistribution from [67]	$\mathcal{O}(\frac{N}{P}(\log N)^3)$	$\mathcal{O}(N)$	$\mathcal{O}(P)$
4.3.3.1.9	Improved Redistribution	$\mathcal{O}(\frac{N}{P}(\log N)^2)$	$\mathcal{O}(N)$	$\mathcal{O}(P)$
4.3.5.1.1	Naïve Redistribution	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(1)$

4.3.3.1.1 Element-wise operations

Perhaps the simplest type of operation to implement in parallel involves applying an element-wise operation[11]. Given a function f and a vector $\mathbf{v}$, the element-wise operation $f \mapsto \mathbf{v}$ applies the function f on every element of the vector such that

$$f \mapsto \mathbf{v} = [f(v_1), f(v_2), \dots, f(v_N)]$$

In our case, normalizing the weights is an example of an element-wise operation. Another example is a vector of *If* operations, $\mathsf{Vif}(\mathbf{a}, \mathbf{b}, \mathbf{c})$ where the ith element in the

[11]Such operations are an example of 'embarrassingly parallel' operations that are arguably trivial to parallelise.

output is b_i if a_i is true and c_i otherwise. Operations involving two inputs and a single output (e.g., element-wise sum or difference) are similarly easy to implement in parallel and involves no data movement between processors.

4.3.3.1.2 Rotation

Another operation involves rotating (with or without wrapping, i.e., cyclic shift) the elements of a vector by a given distance, δ, such that if the input is **a** and the output is **b**, then after the rotation, we have $b(\mod(i+\delta, N)) = a(i)$ where $\mod(x, y)$ is x modulus y. Once again, this algorithmic component is readily parallelised with no data movements between processors.

We will also use partial rotations such that we have a vector of distances, Δ, and not a single 'global' distance, δ. This vector, Δ, has $N' < N$ elements where N' is a power of two. The rotations are then implemented locally to each set of $M = \frac{N}{N'}$ elements. For example if the jth element of Δ is δ_j then $b\left((j-1) \times M + \mod(i+\delta_j, M)\right) = a\left((j-1) \times M + i\right)$ for $1 \leq i \leq M$.

4.3.3.1.3 Sum, max and other commutative operations

The 'adder-tree' can be used to evaluate the sum of a vector of numbers, which are associated with the leaves of the tree. By recursing up the tree, the sum of pairs of numbers is calculated (in parallel across all pairs). The sum of all pairs of pairs of numbers can then be calculated (in parallel across all pairs of pairs). Exemplified in Figure 4.2(a-c), the process repeats until reaching the root node of the tree where the sum of all the numbers is calculated by summing the sum of the two halves of the data (see Figure 4.2(d). In fact, as known since the development of the infamous Array Programming Language (APL) [51], this same approach can be used for any binary operation, $\oplus$, that is commutative such that

$$((a \oplus b) \oplus c) \oplus d = (a \oplus b) \oplus (c \oplus d) \tag{4.1}$$

Relevant examples of operations which can be calculated in this way include the sum, the maximum and minimum, and the first non-zero element of a set of numbers (denoted as $\mathsf{First}(.)$ in Algorithms 10 and 11). For such operations, with N processors processing N data through a binary tree, the time-complexity is the depth of the tree, i.e., $\log_2 N$. Near the bottom of the tree, the total communication required is proportional to the number of processors.

As should be evident, an upside-down version of the same tree can be used to implement an $\mathsf{Expand}(a)$ operation, which involves making all elements of a vector equal to the single value of a.

4.3.3.1.4 Cumulative sum

While using a tree to calculate a sum efficiently is well known, a closely related approach to calculate a cumulative sum[12] efficiently appears to be less well-known by researchers working on particle filters. Of course, a naïve implementation involves computing the cumulative sum by adding each element of the input to the previous element of the output. Such an approach has a run-time of N. However, a more-efficient approach has existed since the development of APL, if not for longer[13].

To impart an intuition as to how this could be possible, the key idea is to exploit the partial sums that are calculated in an adder-tree and to express each element of the cumulative sum as a sum of these (efficiently calculated) partial sums. The process that exploits this insight then involves a second tree in which the values at every level are propagated to the level below, replacing the values that were calculated in the adder-tree. More specifically, through the downward propagation, the value at each parent node is propagated to its right child and left child nodes. The new value for the left child is the difference between the values at the parent and right child nodes (as calculated in the adder-tree). The new value for the right child is the same as the parent node. Figures 4.2(e)-(g) provide an example. With this forward and backward pass of the tree, a cumulative sum is obtained in $2\log_2 N$ steps.

4.3.3.1.5 Normalising the weights

Normalising the weights is an example of an operation that can be implemented using the building blocks described above. The sum is calculated using an adder-tree (as described in Section 4.3.3.1.3), distributed to all the data (Section 4.3.3.1.3) and an element-wise divide (as in Section 4.3.3.1.1) used to calculate the normalised weights.

4.3.3.1.6 Minimum Variance Resampling

Resampling involves determining the number of copies of each particle that are needed. We specifically describe minimum variance resampling, for which the number of copies of the ith particle is

$$m_i = \lfloor C_i \rfloor - \lfloor C_{i-1} \rfloor \tag{4.2}$$

where $\lfloor x \rfloor$ is the floor[14] of x, and

$$C_i = N \sum_{j=1}^{i} w_i + \epsilon \tag{4.3}$$

[12]Note that the cumulative sum is sometimes referred to as a prefix sum, so there is no difference between a prefix sum and a cumulative sum.

[13]APL describes an approach to calculating a sum, maximum or minimum as *reduction* operations. The approach to calculating a cumulative sum is described as a *scan* operation and can be used to calculate, for example, cumulative maximums and minimums. Scan operations take a binary operator $\oplus$ and an N-element vector $\mathbf{a} = [a_1, a_2, \ldots, a_N]$, and return an N-element vector $\mathbf{a}_\oplus = [a_1, (a_1 \oplus a_2), \ldots, (a_1 \oplus a_2 \oplus a_3 \oplus \ldots \oplus a_N)]$. However, here we are only concerned with cumulative sums.

[14]The floor of x is the largest integer smaller than or equal to x.

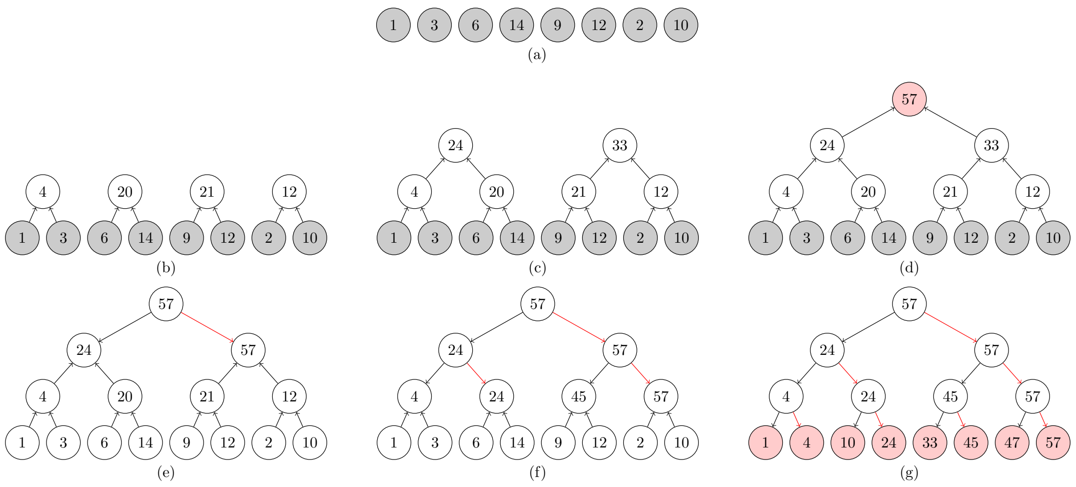


FIGURE 4.2: Example of cumulative sum for N=8 numbers. Subfigures (a)-(d) describe the sum computation, while the remaining balanced binary trees shown in subfigures (e)-(g) describe how the backward pass culminates in calculation of the cumulative sum of the given sequence.

is the cumulative sum and where $\epsilon \sim \left[0, \frac{1}{N}\right]$ and $C_0 = 0$.

Equation 4.2 uses only element-wise operations (Section 4.3.3.1.1) and a rotation (by a single element and as described in section 4.3.3.1.2). Equation 4.3 involves a cumulative sum (Section 4.3.3.1.4) and an addition (Section 4.3.3.1.1). Thus, the building blocks described to this point can be used to implement Equations (4.2) and (4.3).

4.3.3.1.7 Sorting

Quicksort [46] is well known with an average time complexity of $\mathcal{O}(N \log_2 N)$. However, we focus here on the bitonic sort algorithm [16] with a time complexity of $\mathcal{O}(\frac{N}{P}(\log_2 N)^2)$ and spatial complexity of $\mathcal{O}(N)$. The number of data movements at each iteration is P. The reason for this choice is that we want to guarantee the time taken to perform sorting. While it is possible to parallelise quicksort, the ability to do so is data dependent. In contrast, bitonic sort features deterministic time complexity with a balanced load across (up to) N processors.

At the fundamental level, a *bitonic sequence* forms the basis for the bitonic sort. A sequence $\mathbf{a} = [a_1, a_2, \ldots, a_N]$ is a bitonic sequence if $a_1 \leq a_2 \leq \ldots \leq a_k \geq \ldots \geq a_N$ for some k, $1 \leq k \leq N$ or if this condition holds for any rotation of $\mathbf{a}$.

For an intuition as to how the algorithm works, note that at a certain point it the algorithm, we have N data in a bitonic sequence. The first 'half' of the data are sorted in ascending order and the second half are sorted in descending order[15]. Consider the ith element in the first half and the ith element in the second half. There are $\frac{N}{2} - 1$

[15]A similar argument works if the first half are sorted in descending order and the second half are sorted in ascending order.

data between these two elements. They must all be larger than the smallest of the two elements which the data are between. There must therefore be at least $\frac{N}{2}$ data that are larger than the smallest of the two elements. This smallest element must therefore be one of the lowest $\frac{N}{2}$ data (it cannot be one of the largest $\frac{N}{2}$ data if there are at least $\frac{N}{2}$ data larger than it). An upside-down version of the same argument makes clear that the largest of these two elements must be one of the largest $\frac{N}{2}$ data. Finally, it also follows that after this operation, the first $\frac{N}{2}$ data are a bitonic sequence and the second $\frac{N}{2}$ data are a bitonic sequences. Thus, given a bitonic sequence, by comparing all pairs of data that are a distance of $\frac{N}{2}$ apart, and swapping the points if needed, we can ensure all the larger elements are in the first $\frac{N}{2}$ data, which forms a bitonic sequence, and all the smaller elements are in the second $\frac{N}{2}$ data, which also forms a bitonic sequence. We can then apply the same comparison structure on each of the two bitonic (smaller) sequences. This process can be applied recursively until pairs of points are compared, and the data are sorted

This process is known as the 'bitonic merge' and requires $\mathcal{O}(\log_2 N)$ steps (with $\mathcal{O}(N)$ spatial complexity) to convert a bitonic sequence into a sorted sequence. To generate the bitonic sequence needed from arbitrary input data[16], we apply bitonic sort to put the first $\frac{N}{2}$ input data into ascending order and apply bitonic sort again to put the second $\frac{N}{2}$ input data into descending order. An analysis of this recursive use of bitonic sort yields a bitonic sort requiring $\frac{n^2-n}{2}$ iterations where $n = \log_2 N$ and, at every step, the algorithm performs $\frac{N}{2}$ each involving the swapping of two data according to a criterion defined by the position of the comparison in the network. This process can be implemented using the building blocks described in Sections 4.3.3.1.1 and 4.3.3.1.2. A bitonic sort with eight numbers is demonstrated in Figure 4.3.

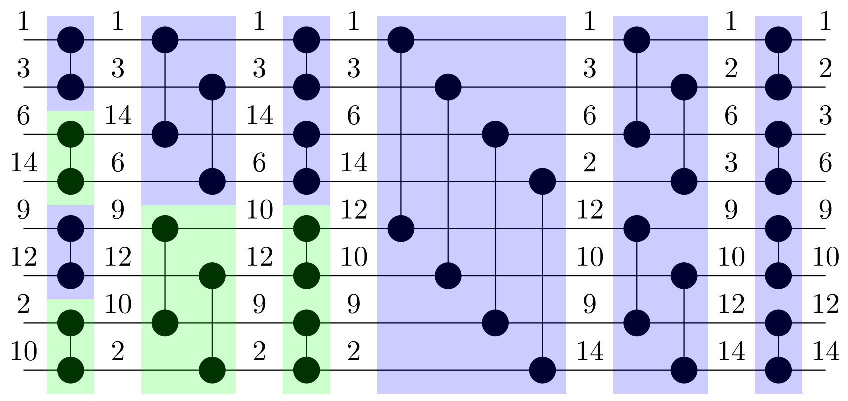
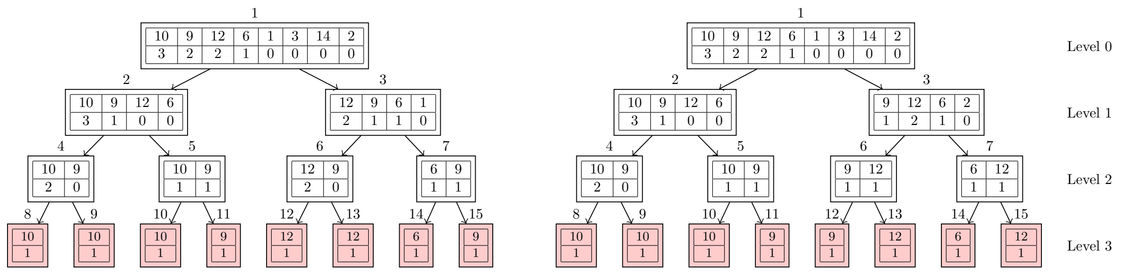


FIGURE 4.3: Example of a bitonic sort using eight numbers. Each horizontal wire corresponds to a core. The blue colour denotes that the larger value will be stored at the lower wire after the comparison, while the green colour represents the opposite scenario.

[16]This process is sometimes known as 'bitonic build'.



(a) Original (b) New

FIGURE 4.4: An example of the redistribution for $\mathbf{x} = [10, 9, 12, 6, 1, 3, 14, 2]$ and $\mathbf{m} = [3, 2, 2, 1, 0, 0, 0, 0]$ using the original and improved (new) redistribute. The original redistribution always sorts the number of copies vector (bottom vector) in descending order, while this is not required in the new redistribution (e.g. see node no. 3).

4.3.3.1.8 Redistribution: Original version

In the original version from [67], the redistribution algorithm takes two inputs, the old population of particles $\mathbf{x}$ and the number of copies $\mathbf{m}$, and produces the new population of particles, $\mathbf{x}^*$, as the output.

In [67], a divide-and-conquer algorithm was described for implementing the redistribute. The procedure involves sorting the particles in decreasing order of the number of copies. With N data, the sum of the elements of $\mathbf{m}$ must be N. The approach is then to divide the data into two smaller datasets, each of which has $\frac{N}{2}$ elements and is such that the corresponding elements of $\mathbf{m}$ are sorted and sum to $\frac{N}{2}$. This can be achieved by finding the *pivot*, which we define as leftmost element in $\mathbf{m}$ for which the associated value of the cumulative sum is $\frac{N}{2}$ or greater. In general, the pivot needs to be split into two constituent parts such that the two smaller datasets can both sum to $\frac{N}{2}$. We refer to these two parts as the left-pivot and right-pivot. The data to the left of the pivot and including the left-pivot can be used to produce one of the two smaller datasets. The right-pivot and the data to the right of the pivot can be used to produce the other of the two smaller datasets. Both smaller datasets are then sorted[17] in decreasing order of m. For the special case when the value of the right-pivot is zero, the rotation needed is one less than otherwise. This procedure can be intuitively considered as operating on a tree. Applying the procedure recursively down the tree, until the leaf nodes are encountered, completes the redistribution. Figure 4.4 illustrates this procedure.

The operation of this algorithm is not dependent on $\mathbf{m}$, and, therefore, also not dependent on the distribution of the weights. Also, the sort can change (somewhat counter-intuitively and seemingly unnecessarily) the order of numbers in a list when elements of the list are not unique. Finally, if no copies of a particle are to be generated, then the identity of the corresponding particle is irrelevant to the eventual output of the algorithm.

[17]The first dataset is actually already sorted, but the second dataset is, in general, not sorted.

The procedure, described in Algorithm 4.3.3.1.1, sum (Section 4.3.3.1.3), cumulative sum (Section 4.3.3.1.4), rotations (Section 4.3.3.1.2) and sort (Section 4.3.3.1.7). Note that the description makes use of three functions (LeftHalf(.), RightHalf(.) and Combine(.)), which are included for clarity and to have zero computational cost. Also, note that the implementation is described in a way that involves recursion. It is possible to 'unwrap' the recursive implementation such that all operations (at all stages in the tree) are implemented on datasets of the same size, N. Doing so is conceptually straightforward though the bookkeeping required is non-trivial.

Algorithm 10 Redistribute: $\mathcal{O}(\frac{N}{P}(\log_2 N)^3)$ implementation.

```
1: Function $\mathbf{x} = \text{Redistribute}(\mathbf{m}, \mathbf{x})$
2: ▷ $\mathbf{m}$: Number of copies (sorted in descending order)
3: ▷ $\mathbf{x}$: Particles
4: if $\text{Length}(\mathbf{m}) > 1$ then
5:     ▷ Calculate Cumulative Sum
6:     $\mathbf{c} \leftarrow \text{CumSum}(\mathbf{m})$
7:     ▷ Identify Pivot
8:     $i_p \leftarrow \text{First}(\mathbf{c} \geq \frac{N}{2})$
9:     $\mathbf{p} \leftarrow \text{Expand}(i_p)$
10:    ▷ Calculate Left-Pivot and Right-Pivot
11:    ▷ $\mathbf{i}$ simply indexes the elements of $\mathbf{m}$ and $\mathbf{0}$ is a vector of zeros
12:    $\mathbf{lp} \leftarrow \text{Vif}(\mathbf{i} = \mathbf{p}, \mathbf{c} - \frac{N}{2}, \mathbf{0})$
13:    $\mathbf{rp} \leftarrow \text{Vif}(\mathbf{i} = \mathbf{p}, \frac{N}{2} - \text{Rotate}(\mathbf{c}, 1), \mathbf{0})$
14:    ▷ Generate Smaller Datasets
15:    $\mathbf{l} \leftarrow \text{LeftHalf}(\text{Vif}(\mathbf{i} < \mathbf{p}, \mathbf{m}, \mathbf{lp}))$
16:    $\mathbf{lx} \leftarrow \text{LeftHalf}(\mathbf{x})$
17:    $\mathbf{r} \leftarrow \text{Vif}(\mathbf{i} > \mathbf{p}, \mathbf{m}, \mathbf{rp})$
18:    ▷ Calculate Rotation of $\mathbf{r}$
19:    $inc \leftarrow \text{Sum}(\text{Vif}(\mathbf{c} = \frac{N}{2}, \mathbf{1}, \mathbf{0}))$
20:    $\mathbf{r} \leftarrow \text{RightHalf}(\text{Rotate}(\mathbf{r}, i_p + inc))$
21:    $\mathbf{rx} \leftarrow \text{RightHalf}(\text{Rotate}(\mathbf{x}, i_p + inc))$
22:    ▷ Sort Right Half
23:    $\mathbf{r} \leftarrow \text{Sort}(\mathbf{r})$
24:    ▷ Divide-and-conquer
25:    $\mathbf{lx} \leftarrow \text{Redistribute}(\mathbf{l}, \mathbf{lx})$
26:    $\mathbf{rx} \leftarrow \text{Redistribute}(\mathbf{r}, \mathbf{rx})$
27:    ▷ Combine Outputs
28:    $\mathbf{x} \leftarrow \text{Combine}(\mathbf{lx}, \mathbf{rx})$
29: end if
30: EndFunction
```

The time complexity of this redistribution algorithm $\mathcal{O}(\frac{N}{P}(\log_2 N)^3)$ in parallel with N processors since a (bitonic) sort (with complexity of $\mathcal{O}(\frac{N}{P}(\log_2 N)^2)$) is used at each stage in the divide-and-conquer. This analysis contradicts the (erroneous) claim in [67] that the time complexity of this algorithm is $\mathcal{O}(\frac{N}{P}(\log_2 N)^2)$. Again, the communication complexity is P.

4.3.3.1.9 Redistribution: Improved version

The redistribution algorithm described in Section 4.3.3.1.8 is a divide-and-conquer algorithm that ensures that, at each node in the tree, $\mathbf{m}$ sums to its length, N, and is sorted. The sorting is sufficient to ensure that rotation can be used to replace some of the (right-most) zeros with the (right-most) non-zero elements of $\mathbf{m}$ that sum to $\frac{N}{2}$.

Here, we exploit the observation that it is possible to define an alternative divide-and-conquer strategy. More specifically, we ensure that, at each node in the tree, $\mathbf{m}$ sums to its length, N, and has all its non-zero values to the left of all values that are zero. Since such a sequence only has trailing zeros, we call it an All-Trailing-Zeros (ATZ) sequence[18]. While a sort is sufficient to generate an ATZ sequence, it is easier to generate an ATZ sequence than a sorted sequence, as we will demonstrate.

The new algorithm, at each node in the tree, starts with $\mathbf{m}$, which sums to its length, N, and is an ATZ sequence. To proceed, as previously, we find the pivot (as defined in Section 4.3.3.1.8). As previously, the data to the left of the pivot and the left-pivot can be used to produce one of the two smaller datasets. However, in contrast to the approach described in Section 4.3.3.1.8, we can simply use the right-pivot and the data to the right of the pivot to generate the second smaller dataset (without any need for sort). Both these smaller datasets then sum to $\frac{N}{2}$ and are ATZ sequences. Note that, as with the approach described in Section 4.3.3.1.8, there is a special case that occurs when the value of the right-pivot is zero.

We need to generate an ATZ sequence. To achieve this, we propose to use (bitonic) sort (once). After this initial sort, the procedure can be described using element-wise operations (as in Section 4.3.3.1.1), sum (Section 4.3.3.1.3), cumulative sum (Section 4.3.3.1.4) and rotations (Section 4.3.3.1.2). We emphasise that there is no need for a sort after the initial generation of an ATZ sequence. As a result, while the algorithm described in Section 4.3.3.1.8 has time-complexity of $\mathcal{O}(\frac{N}{P}(\log_2 N)^3)$, the time complexity of the algorithm described in this section is $\mathcal{O}(\frac{N}{P}(\log_2 N)^2)$. Notice that the number of data movements is still P. Algorithm 11 provides a description of this algorithm, which has a strong similarity to Algorithm 10. Once again, it is possible to 'unwrap' the recursive implementation albeit with some non-trivial bookkeeping.

4.3.4 Mapping particle filtering into MapReduce

The descriptions provided in Section 4.3.3.1 describe distributed operations that can manipulate vectors (albeit after some unwrapping of the recursive descriptions). As discussed in Section 4.3.2.2, the fundamental notion of MapReduce is the processing of (key, value) pairs. In the context of particle filtering, none of the properties of the particles (weight or state) qualifies to be a key. However, we can give each particle a unique index and use this index as the key by considering the particles as being a set

[18] We suspect such a sequence may have a name identified in the literature that we are not aware. However, in this context, we adopt an intuitive name for clarity.

Algorithm 11 Redistribute: $\mathcal{O}(\frac{N}{P}(\log_2 N)^2)$ implementation.

```
1:  Function $\mathbf{x} = \mathsf{Redistribute}(\mathbf{m}, \mathbf{x})$
2:  ▷ $\mathbf{m}$: Number of copies (in an ATZ sequence)
3:  ▷ $\mathbf{x}$: Particles
4:  if $\mathsf{Length}(\mathbf{m}) > 1$ then
5:      ▷ Calculate Cumulative Sum
6:      $\mathbf{c} \leftarrow \mathsf{CumSum}(\mathbf{m})$
7:      ▷ Identify Pivot
8:      $i_p \leftarrow \mathsf{First}(\mathbf{c} \geq \frac{N}{2})$)
9:      $\mathbf{p} \leftarrow \mathsf{Expand}(i_p)$
10:     ▷ Calculate Left-Pivot and Right-Pivot
11:     ▷ $\mathbf{i}$ simply indexes the elements of $\mathbf{m}$ and $\mathbf{0}$ is a vector of zeros
12:     $\mathbf{lp} \leftarrow \mathsf{Vif}(\mathbf{i} = \mathbf{p}, \mathbf{c} - \frac{N}{2}, \mathbf{0})$
13:     $\mathbf{rp} \leftarrow \mathsf{Vif}(\mathbf{i} = \mathbf{p}, \frac{N}{2} - \mathsf{Rotate}(\mathbf{c}, 1), \mathbf{0})$
14:     ▷ Generate Smaller Datasets
15:     $\mathbf{l} \leftarrow \mathsf{LeftHalf}(\mathsf{Vif}(\mathbf{i} < \mathbf{p}, \mathbf{m}, \mathbf{lp}))$
16:     $\mathbf{lx} \leftarrow \mathsf{LeftHalf}(\mathbf{x})$
17:     $\mathbf{r} \leftarrow \mathsf{Vif}(\mathbf{i} > \mathbf{p}, \mathbf{m}, \mathbf{rp})$
18:     ▷ Calculate Rotation of $\mathbf{r}$
19:     $inc \leftarrow \mathsf{Sum}(\mathsf{Vif}(\mathbf{c} = \frac{N}{2}, \mathbf{1}, \mathbf{0}))$
20:     $\mathbf{r} \leftarrow \mathsf{RightHalf}(\mathsf{Rotate}(\mathbf{r}, i_p + inc))$
21:     $\mathbf{rx} \leftarrow \mathsf{RightHalf}(\mathsf{Rotate}(\mathbf{x}, i_p + inc))$
22:     ▷ Divide-and-conquer
23:     $\mathbf{lx} \leftarrow \mathsf{Redistribute}(\mathbf{l}, \mathbf{lx})$
24:     $\mathbf{rx} \leftarrow \mathsf{Redistribute}(\mathbf{r}, \mathbf{rx})$
25:     ▷ Combine Outputs
26:     $\mathbf{x} \leftarrow \mathsf{Combine}(\mathbf{lx}, \mathbf{rx})$
27: end if
28: EndFunction
```

Table 4.3: Details of the Experimental Platform used for Evaluation.

Details	**Single Node System**	**Multi-Node System**
Name	Platform 1	Platform 2
Number of Nodes	1	28
Hardware Cores	16	512
Operating System	Linux	IBM Unix
Primary Memory	16GB	384GB
Spark Version	1.6.2	1.4.1
Hadoop Version	2.7.2	2.7.1

$\{i, x_i, w_i\}$ where $i \in \{1, \ldots, N\}$ and, as previously, N is the number of particles, x_i is the state and w_i is the corresponding weight of the ith particle.

4.3.5 Evaluation

An extensive evaluation of our algorithm on two different systems is provided in Table 4.3. The evaluation process included the algorithms outlined in Section 4.3.3.1 on

Hadoop and Spark, the two key frameworks that support MapReduce and which were mentioned in Section 4.3.2. We used the standard estimation problem involving a scalar state and a computationally inexpensive proposal, likelihood, and dynamic model that is widely used in the particle filtering community [12]. This scenario emphasises the need for efficient resampling since, as is often the case, the likelihood, dynamics, and proposal are computationally demanding, and the relative merits of different resampling schemes would be less apparent. Our evaluation focuses on specific aspects of the implementation described as follows:

1. In Section 4.3.5.1, we start by providing evidence that, in contrast to a naïve implementation, the particle filter we developed exploits multi-core architectures while having deterministic run-time.
2. In Section 4.3.5.2, as a precursor to a detailed evaluation and analysis, we analyse the overall profile of the particle filtering algorithm for implementations on a single core, using Hadoop and Spark.
3. In Section 4.3.5.3, for both the Spark and Hadoop implementations, we compare the performance of our new algorithms relative to a single mapper and a single reducer. In doing so, we compare the overall performance as well as the fundamental building blocks of the particle filtering algorithm. This section provides a thorough understanding of these algorithms' performance on these two key frameworks that support MapReduce.
4. Given that the Spark implementation (unsurprisingly) outperforms the Hadoop implementation, we focus on the Spark implementation. In Section 4.3.5.4, we compare the two versions of the redistribution algorithm described in Sections 4.3.3.1.8 and 4.3.3.1.9 as a function of the number of particles and cores. The intent is that this detailed comparison provides insight into the performance that is achievable using the original and proposed variants of the redistribution algorithm.
5. Finally, in Section 4.3.5.5, we perform a detailed analysis on the speedup and scalability of the redistribution and the overall particle filter.

In performing these evaluations, we identified a basic parameter useful in assessing the algorithmic performance called the Particles Processed per Second (PPS) number, which is the capability to process large amounts of data and directly translates to the number particles that can be processed per unit time.

4.3.5.1 Worst case runtime performance

4.3.5.1.1 Baseline redistribution algorithm

We compare performance against a naïve baseline implementation of the redistribution component, which involves calculation (in parallel) of a cumulative sum of the number of copies. Once this cumulative sum is calculated for each particle in the old population, and each element of the sum communicated to be processed along with its neighbour, we know the first and last indices of particles in the new population that will be copies of this particle in the old population. Then, by performing a loop across the particles in the old population, we populate the new generation of particles (Algorithm 12). In MapReduce, a map function is used for the outer for loop, and within each map, we iterate as many times as needed according to the number of copied elements.

Algorithm 12 Redistribute: $\mathcal{O}(N)$ implementation.

```
 1: Function $\mathbf{x}^* = \mathsf{Redistribute}(\mathbf{m}, \mathbf{x})$
 2: ▷ $\mathbf{m}$: Number of copies
 3: ▷ $\mathbf{x}$: Particles
 4: ▷ $\mathbf{x}^*$: New population of Particles
 5: $\mathbf{i} \leftarrow \mathbf{0}$
 6: for $j = 0 : N$ do
 7:     for $k = 0 : m[j]$ do
 8:         ▷New Population of Particles
 9:         $\mathbf{x}^*[\mathbf{i}] \leftarrow \mathbf{x}[\mathbf{j}]$
10:         $\mathbf{i} \leftarrow \mathbf{i} + \mathbf{1}$
11:     end for
12: end for
13: EndFunction
```

This algorithm, when running across multiple cores, can be expected to have a runtime complexity dependent on the data. To clarify this result, we consider the worst case where the redistribution involves making N (denotes the number of particles) copies of the i^{th} particle (and zero copies of all other particles). In this case, only one core will populate the new generation of particles.

4.3.5.1.2 Runtime performance and variability

We investigate the worst-case performance of a naïve parallel implementation of the redistribution component and compare with our proposed implementation (using Spark). The results are shown in Figures 4.5 and 4.6 for the worst-case where the new population of particles are all copies of a single member from the old population. It should be evident that as the number of cores increase, the runtime of the proposed nearly never increases[19]. In contrast, while the runtime of the naïve implementation initially decreases as the number of cores is increased, it then increases (i.e., such that it is faster in absolute terms to use 8 not 16 cores with Platform 1 and such that it is faster to use less than 50 cores not 512 cores with Platform 2). The reason for this decrease is that the MapReduce framework can use the extra cores to more rapidly process the (many) zeros in the vector describing the number of copies. The subsequent increase in processing time is due to

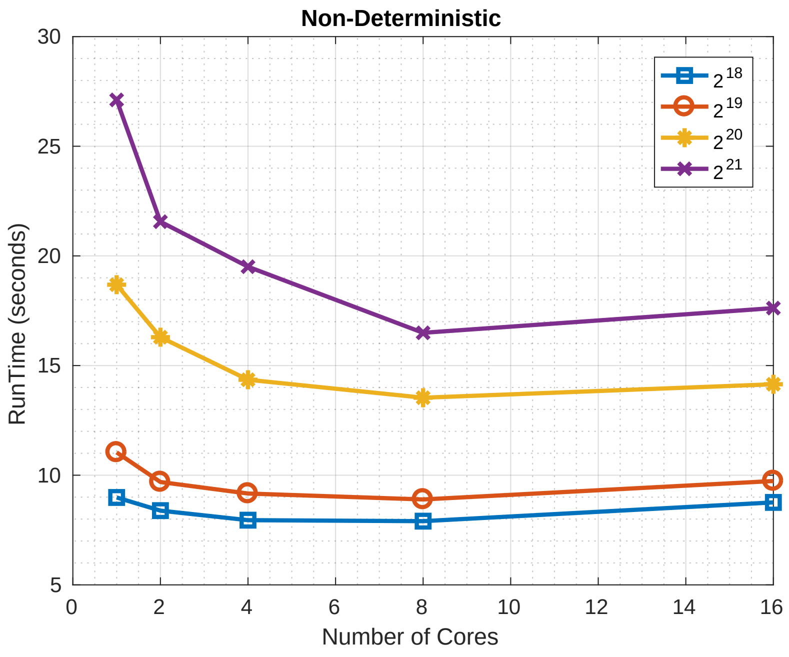
[19]In subsequent sections, we will investigate how and when the decrease in runtime occurs in more detail.

the additional overhead of having multiple cores becomes increasingly significant if only one of the cores is doing the majority of the processing.

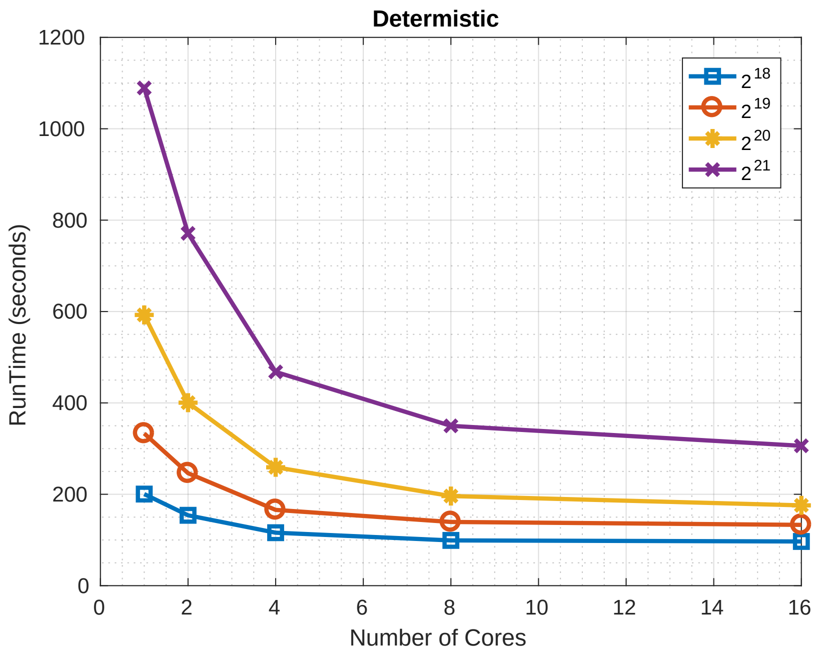
It should also be evident that the absolute runtime (on these platforms and our current Spark implementation) of the deterministic and non-deterministic variants differ significantly such that the naïve implementation can be approximately 20 times faster in the context of both platforms. This is disappointing and motivates future work to refine our initial implementation. However, we presume there are applications where a slower but deterministic runtime is preferable to a faster but data-dependent runtime. In the context of such applications, particularly given the scope to improve the implementation, we perceive our algorithm, if not our current implementation, has utility.

To assess the variation in runtime we experience when considering different distributions of the weights, we compare the performance in the context of the worst-case scenario with that of the best-case scenario[20]. Figure 4.7 describes the average runtime as well as the minimum and maximum runtimes over five runs. It is clear that the fluctuations between the runs are smaller for the deterministic compared to the naïve non-deterministic algorithm. What is less clear, but still discernible, is that the average runtime for the deterministic redistribute is impacted less between the worst- and best-cases than the naïve non-deterministic redistribute. We believe this modest difference points to the runtime being dominated by considerations other than the algorithmic choice, such as MapReduce's overheads, which are common to both algorithms' implementations.

[20]With N particles, the best-case involves replicating each particle exactly once.

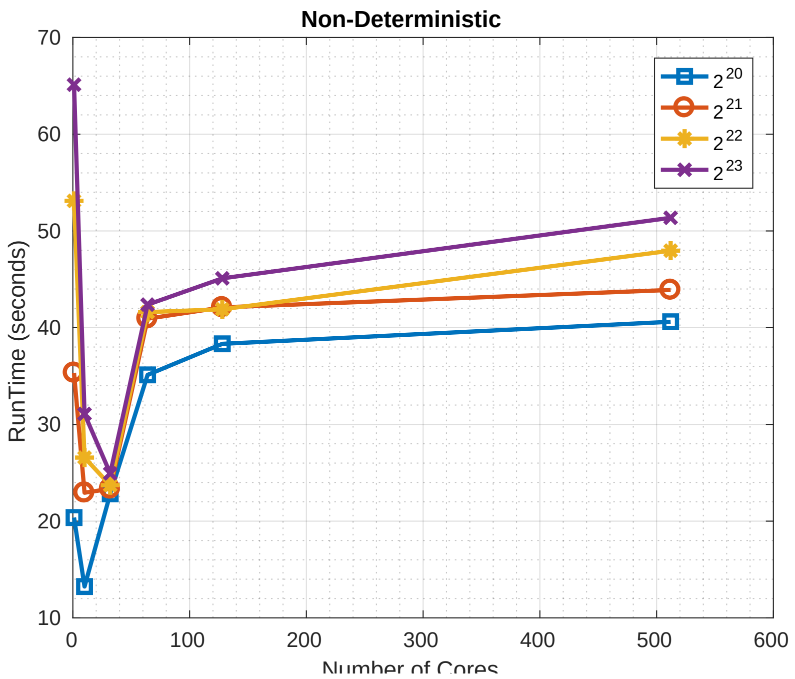


(a) Naïve implementation

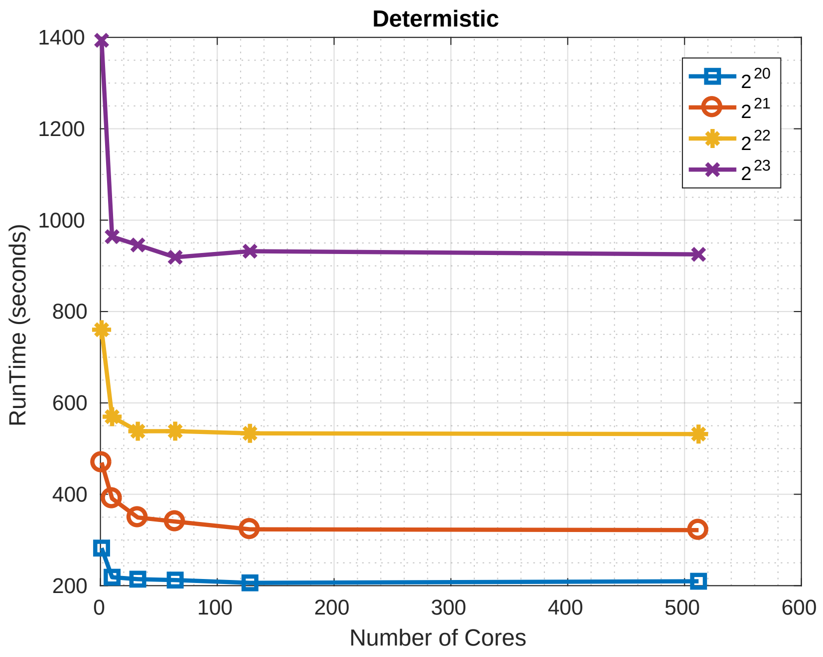


(b) Proposed approach

FIGURE 4.5: Worst-case performance of Redistribution: Platform 1.



(a) Naïve implementation



(b) Proposed approach

FIGURE 4.6: Worst-case performance of Redistribution: Platform 2.

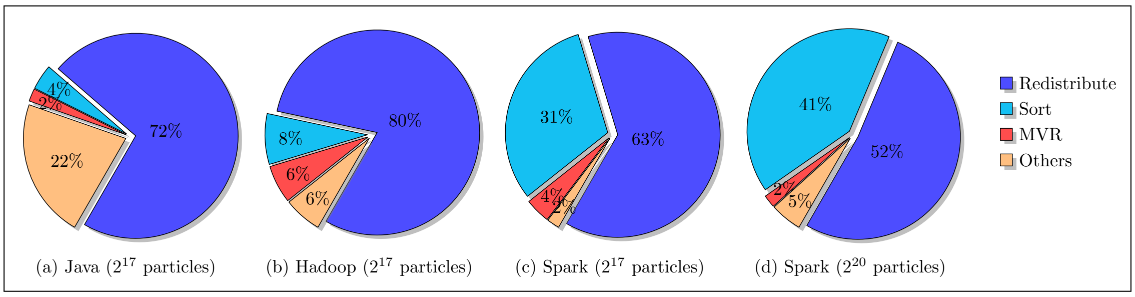


FIGURE 4.8: Overall runtime profile of the particle filtering algorithm for the following implementations: (a) Sequential; (b) Hadoop; (c) Spark with 2^{17} particles; (d) Spark with 2^{20} particles.

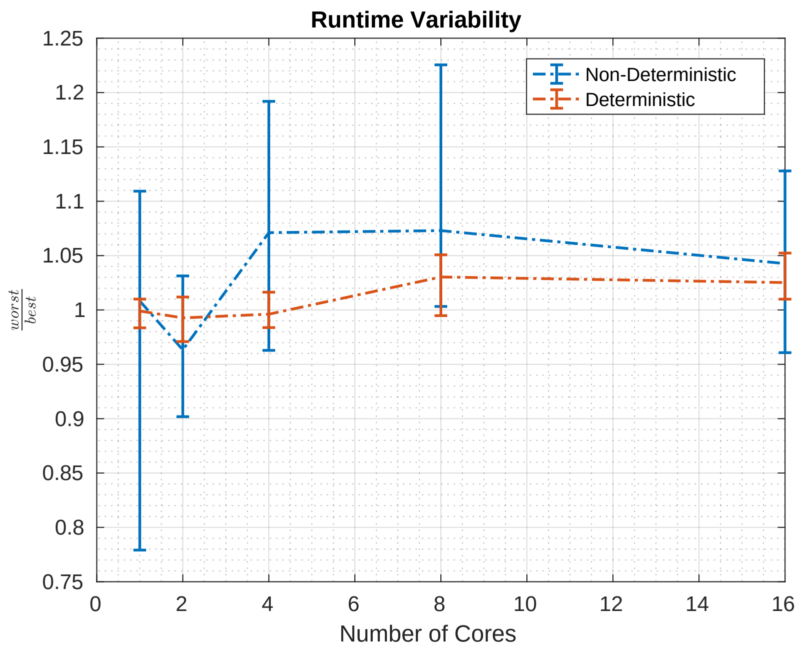


FIGURE 4.7: Ratio of average (and minimum and maximum) run-times for worst-case and best-case scenarios using the deterministic and naïve redistribute.

4.3.5.2 Overall profile

We next compare the performance of a particle filter with the three implementations of a sequential implementation in Java using quicksort in place of the bitonic sort, Hadoop, and Spark. All implementations involve a single core and Platform 1. Figure 4.8 shows the proportion of the runtime associated with redistribution, sort, Minimum Variance Resampling (MVR), and the remaining components (e.g., sum, cumulative sum, diff, and scaling).

As observed in Figure 4.8, most of the time is devoted to the redistribution component. For the Spark implementation, a significant fraction of the remaining time is spent on the sorting component and the fraction of time devoted to redistribution and sorting increases as the number of particles increases.

4.3.5.3 Comparison of Hadoop and Spark

Here, we investigate how the choice of middleware impacts performance in the context of the components of the algorithm and the entire particle filter algorithm. All implementations involve a single core and Platform 1.

4.3.5.3.1 Sum and Cumulative Sum

Figures 4.9 4.10 shows the comparative performance of the sum and cumulative sum components in the Hadoop and Spark frameworks.

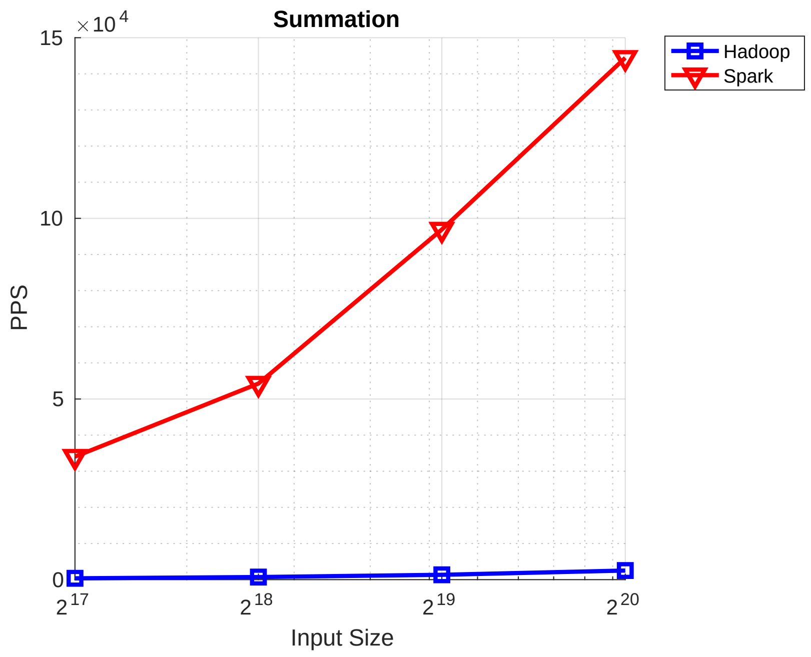


FIGURE 4.9: Summation on Spark and Hadoop.

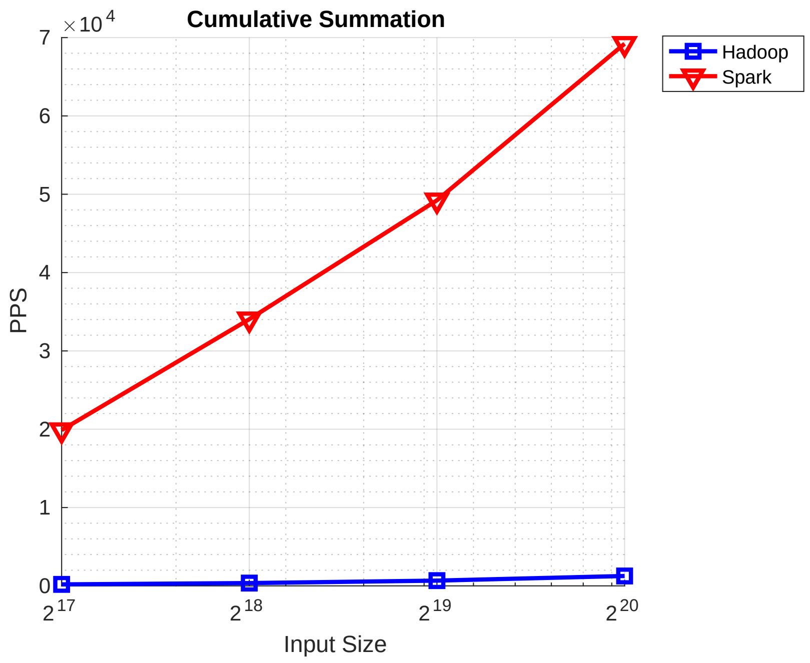


FIGURE 4.10: Cumulative Summation on Spark and Hadoop.

With respect to the PPS parameter, the performance using Spark is superior to that achieved using Hadoop. This stems from the issues discussed in Section 4.3.2 as Spark uses RDDs to makes use of memory (and lazy evaluation) and Hadoop only uses the file system (HDFS) to transfer data from the output of one operation to the input of the next.

It is apparent in both frameworks, especially in the context of Spark, that as the number of particles increases, the number of particles processed per second also increases. This behaviour is because, with more particles, the overheads associated with setting up and tearing down the mappers and reducers are increasingly offset by the parallel operations that make use of these mappers and reducers. The limited extent to which this effect is observed in the context of Hadoop highlights that the overheads associated with opening files in HDFS are significant.

As explained in Section 4.3.3.1, since calculating a summation involves one adder-tree and a cumulative sum involves two, we should expect the PPS for the cumulative sum to be approximately half of that for the summation. A comparison of the two graphs in Figures 4.9 4.10 makes clear that this is approximately the case for both frameworks and all input sizes.

4.3.5.3.2 Bitonic sort and Minimum Variance Resampling

Figures 4.11 4.12 shows the performance for two independent components, *bitonic sort* and *minimum variance resampling*. The performance of *minimum variance resampling* is relatively close to the performance of the cumulative sum (see Figures 4.9 4.10). This is expected since, as explained in Section 4.3.3.1, *minimum variance resampling* includes a cumulative sum.

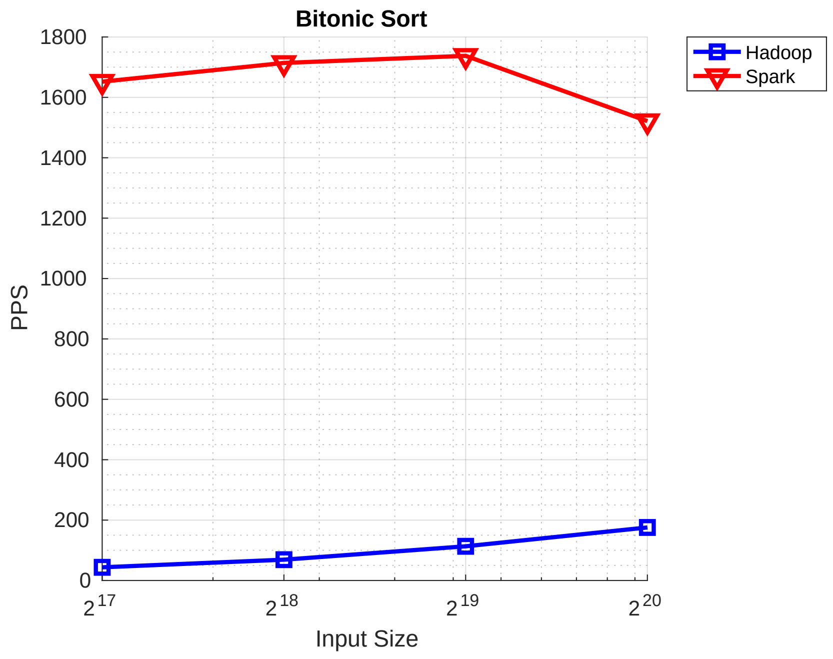
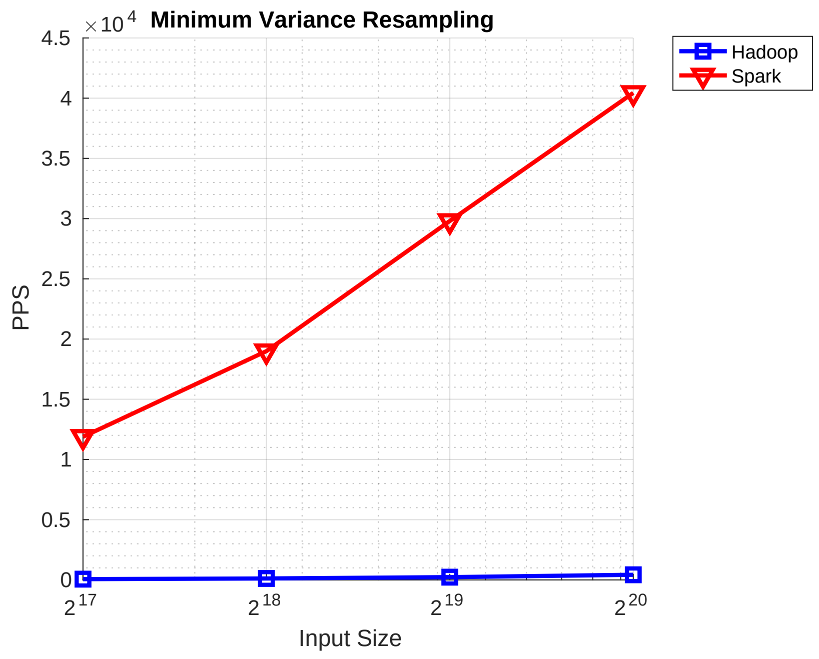


FIGURE 4.11: Bitonic Sort on Spark and Hadoop.



(b) Minimum Variance Resampling

FIGURE 4.12: Minimum variance resampling on Spark and Hadoop.

We again notice, for the same reasons discussed in Section 4.3.5.3.1, the difference in performance between the Spark and Hadoop implementations. As before, for the *minimum variance resampling*, the PPS parameter increases with the number of particles. However, for bitonic sort with Spark, the PPS decreases for large numbers of particles. On investigating further, we observe the *lineages* used to facilitate the lazy evaluation in

Spark[21] become very large with a large number of particles, and appears to cause Spark to become less efficient.

4.3.5.3.3 Redistribution and overall performance

Finally, Figures 4.13 4.14 shows the comparative performance of the redistribution algorithm (as described in Algorithm 11) and the overall particle filtering algorithm.

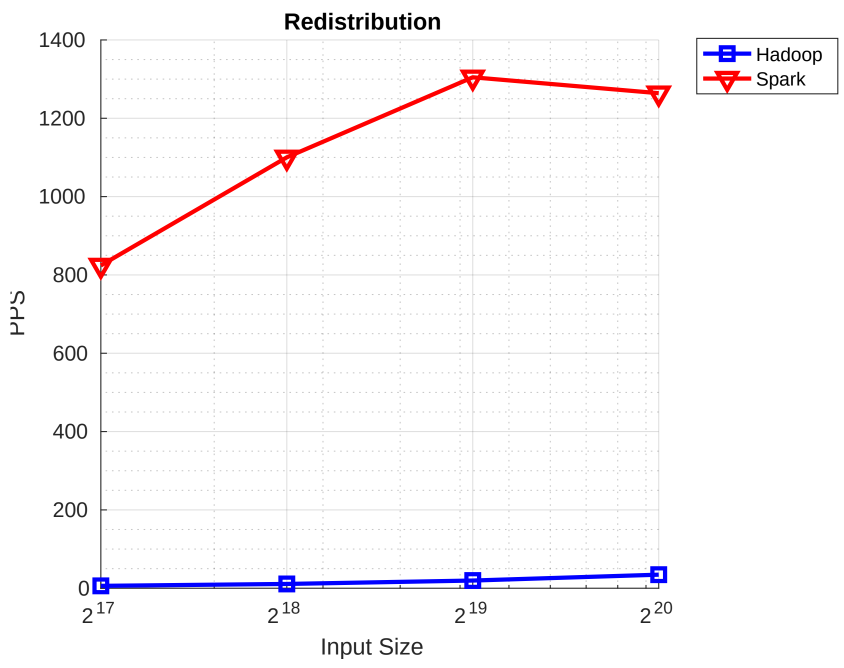


FIGURE 4.13: Redistribution on Spark and Hadoop.

[21]Since Hadoop does not attempt lazy evaluation or use such lineages for another purpose, the same phenomenon is not observed in the context of Hadoop.

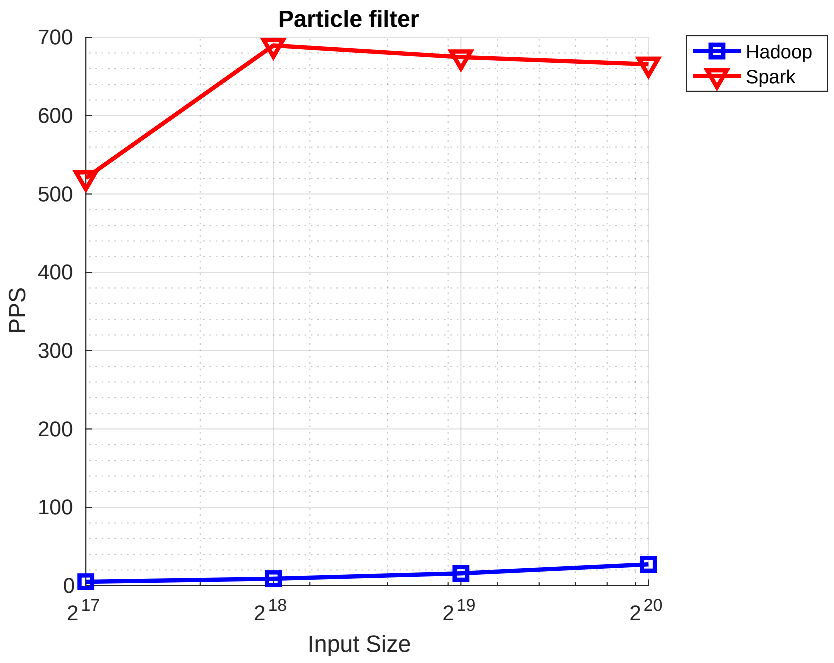


FIGURE 4.14: Overall Particle Filtering on Spark and Hadoop.

Again, we notice the differences between Hadoop and Spark. In the context of the overall particle filter and the largest number of particles considered, these differences are obvious in Spark, relative to Hadoop, offering a considerable speedup of approximately 25-fold[22].

The overall performance of the particle filtering algorithm, when implemented in Spark, decreases for large numbers of particles. Again, on investigation, this appears to be caused by large lineages associated with a large number of particles. Finally, the bitonic sort and redistribution components appear to be limiting the number of particles per second that can be processed by the overall particle filtering algorithm.

4.3.5.4 Impact of using multiple cores

Next, we focus on the Spark implementation (with Platform 1) and compare the performance of the two variants of the redistribution component in isolation and in the context of the overall performance of a particle filter. Specifically, we investigate how performance scales with the number of cores and the number of particles.

4.3.5.4.1 Redistribution component in isolation

Figures 4.15 4.16 compares the performance of the two versions of the redistribution component as a function of the number of particles and number of cores.

[22]In the particle filter the resampling is executed in every iteration. Thus the aforementioned figures correspond to a worst-case speedup.

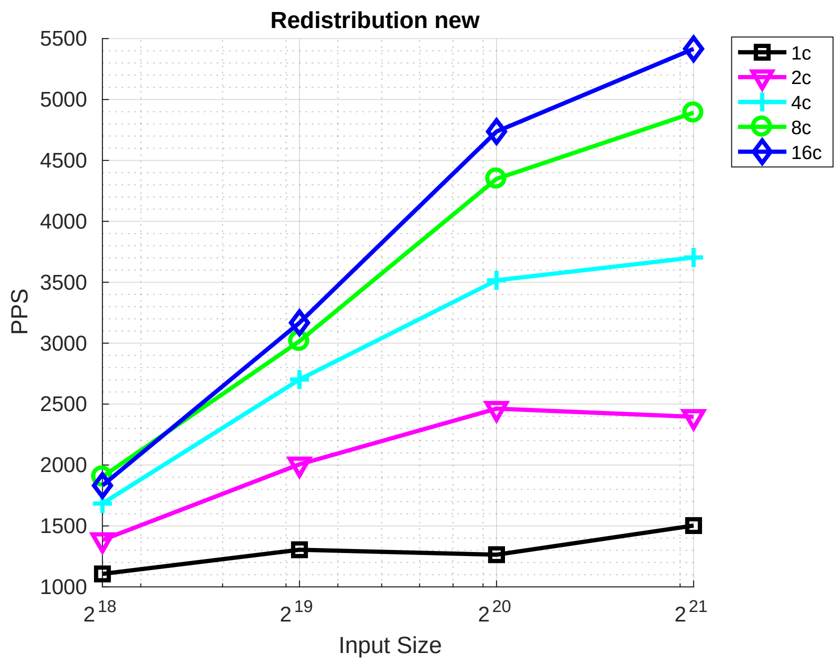


FIGURE 4.15: Performance of the $\mathcal{O}(\frac{N}{P}(\log_2 N)^2)$ Redistribution Component (using Spark).

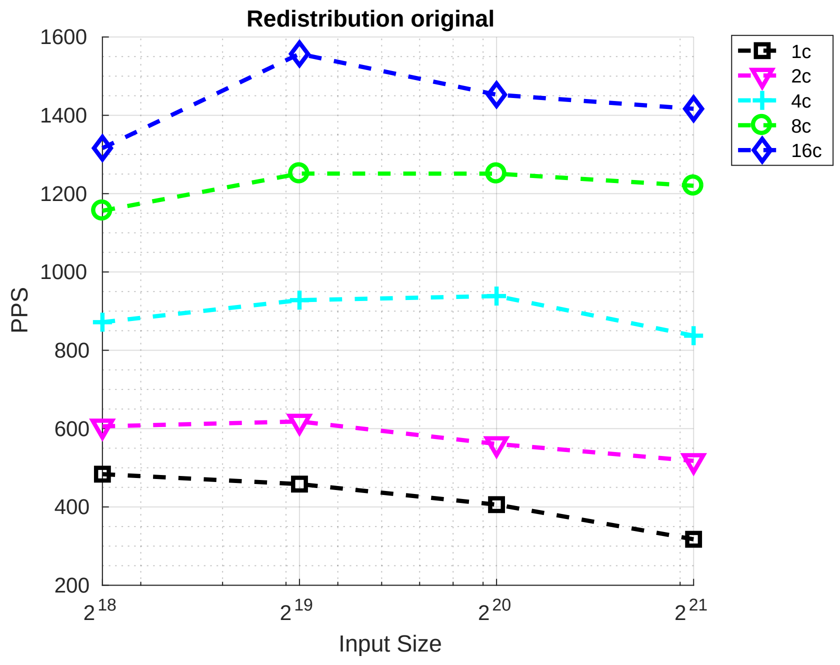


FIGURE 4.16: Performance of the $\mathcal{O}(\frac{N}{P}(\log_2 N)^3)$ Redistribution Component (using Spark).

On a core-to-core basis, the $\mathcal{O}((\log_2 N)^2)$ redistribution component outperforms the $\mathcal{O}((\log_2 N)^3)$ component across all numbers of particles by a margin of up to a factor of approximately 4 (for 16 cores). For all numbers of particles, increasing the number of cores improves performance for both variants of the redistribution component. However, in the context of both variants, the improvement in performance when considered as a ratio is less than the ratio of the number of cores.

In the context of the $\mathcal{O}(\frac{N}{P}\,(\log_2 N)^3)$ variant, increasing the number of particles for a fixed number of cores can significantly reduce the number of particles processed per second. This is not the case for the $\mathcal{O}(\frac{N}{P}\,(\log_2 N)^2)$ variant. Also for the $\mathcal{O}(\frac{N}{P}\,(\log_2 N)^2)$ variant, increasing the number of particles while keeping the number of cores constant improves the number of particles processed per second. However, in the context of the $\mathcal{O}(\frac{N}{P}\,(\log_2 N)^3)$ variant, increasing the number of particles for a fixed number of cores reduces the number of particles processed per second.

4.3.5.4.2 Resulting overall particle filter performance

Figures 4.17 4.18 compares the performance of the original particle filtering algorithm when using the two variants of the redistribution component.

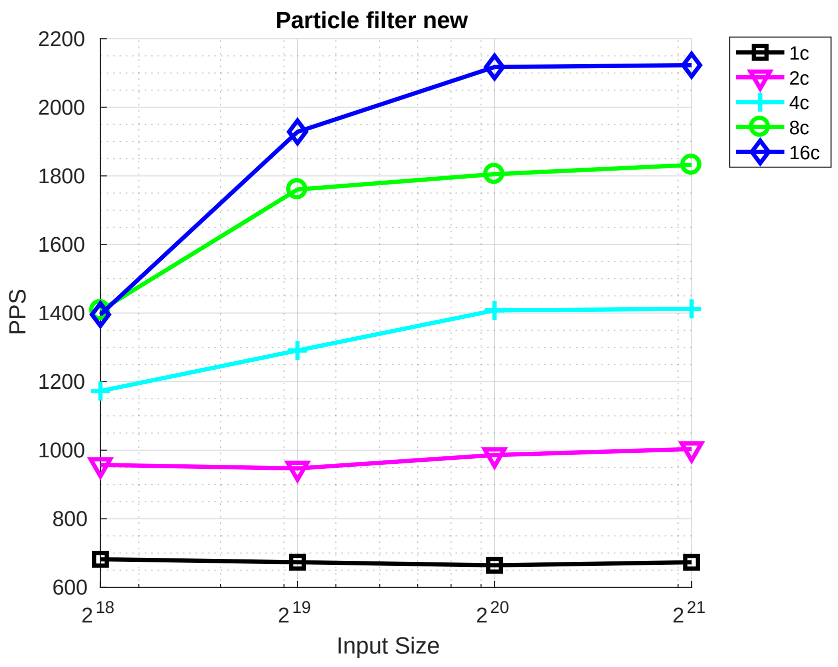


Figure 4.17: Performance of the overall particle filter using the $\mathcal{O}(\frac{N}{P}(\log_2 N)^2)$ redistribution component.

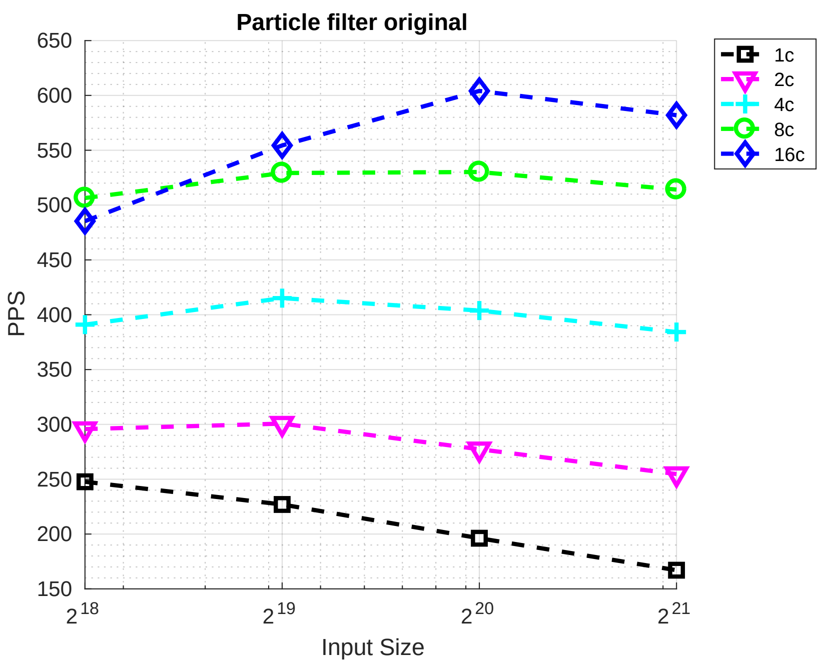


FIGURE 4.18: Performance of the overall particle filter using the $\mathcal{O}(\frac{N}{P}(\log_2 N)^3)$ redistribution component.

The comparative performance observed in the context of the redistribution component in isolation is also evident when comparing the performance of the overall particle filter. Indeed, the use of the $\mathcal{O}(\frac{N}{P}(\log_2 N)^2)$ variant of the redistribution results in approximately a fourfold increase in the number of particles processed per second. The trends observed in the context of the redistribution component in isolation are also apparent in the context of the overall particle filter.

4.3.5.5 Speedup and scalability analysis

We now focus on the speedup that the $\mathcal{O}(\frac{N}{P}(\log_2 N)^2)$ variant of the redistribution component offers relative to the $\mathcal{O}(\frac{N}{P}(\log_2 N)^3)$ variant and the scalability of the $\mathcal{O}(\frac{N}{P}(\log_2 N)^2)$ variant, i.e., the extent to which using more cores improves performance.

We quantify speedup as the ratio of the number of particles per second for a fixed number of particles and number of cores. We quantify scalability, in the context of a fixed number of particles[23], as the ratio of the number of particles per second with N cores relative to the number of particles per second with a single core. We compare performance in the context of both platforms for different numbers of particles.

4.3.5.5.1 Redistribution component in isolation

Figures 4.19 4.20 and 4.21 4.22 describe the speedup and scalability of the $\mathcal{O}(\frac{N}{P}(\log_2 N)^2)$ redistribution component in the context of platforms 1 and 2 respectively.

[23]Since the problem size remains fixed, we are actually quantifying *strong scaling* [43].

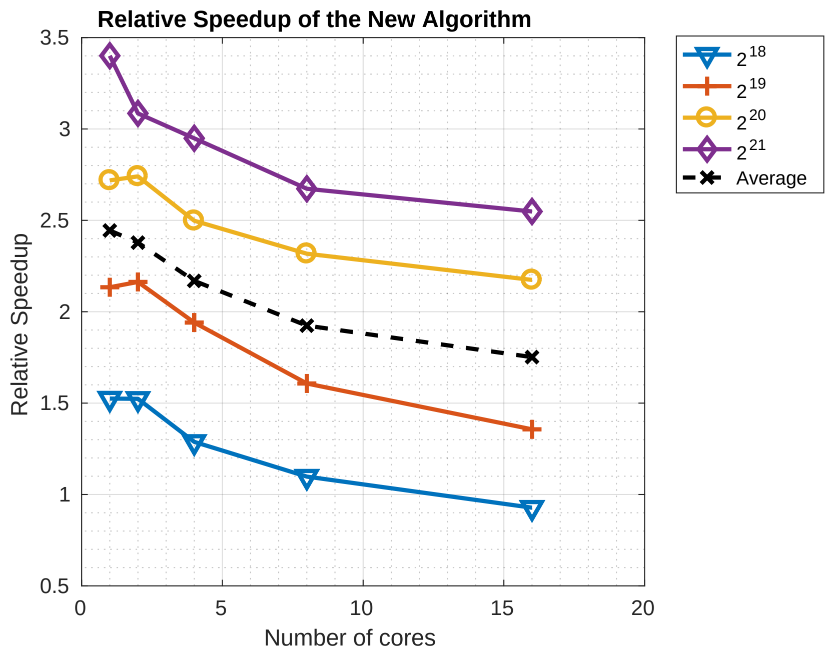


FIGURE 4.19: Relative Speedup $\mathcal{O}(\frac{N}{P}(\log_2 N)^2)$ variant of the Redistribution component on Platform 1.

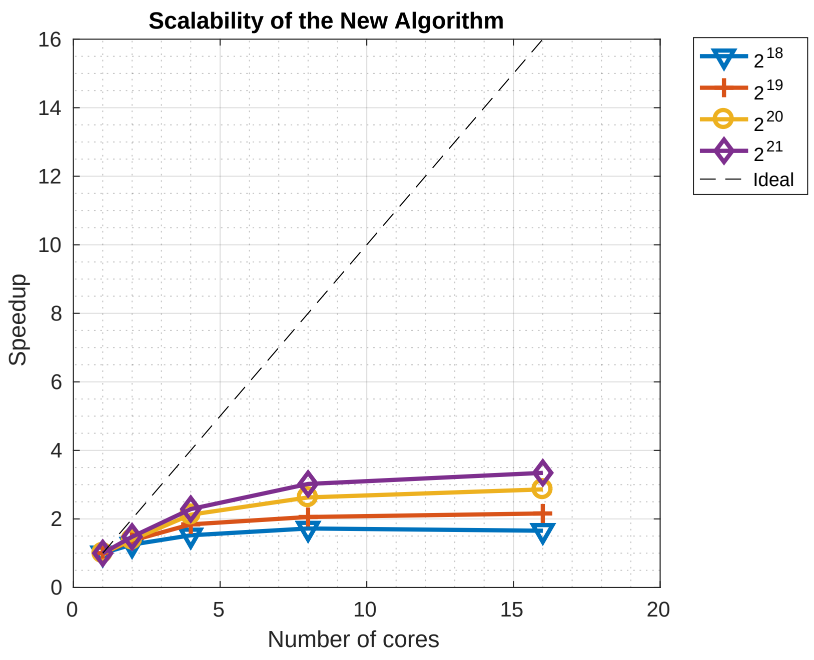


FIGURE 4.20: Scalability of the $\mathcal{O}(\frac{N}{P}(\log_2 N)^2)$ variant of the Redistribution component on Platform 1.

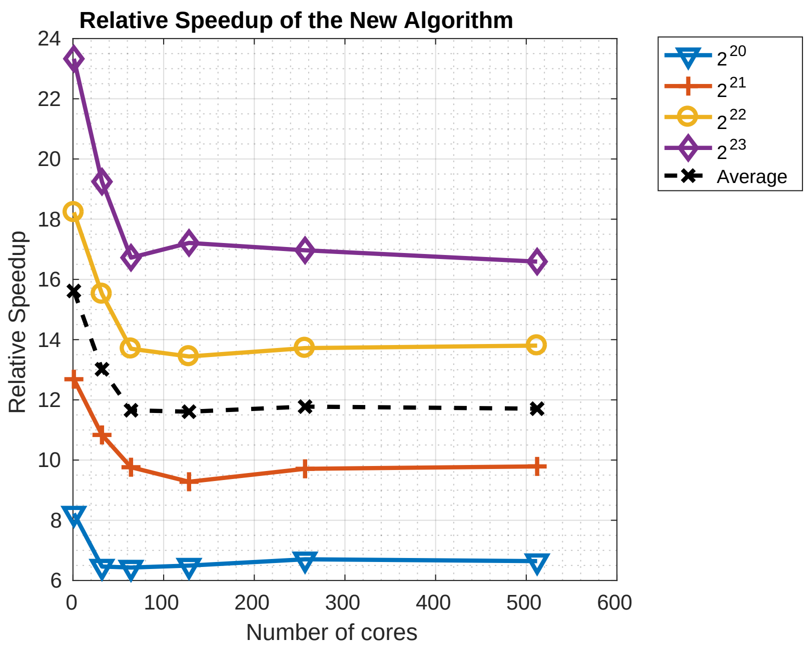


FIGURE 4.21: Relative Speedup of the $\mathcal{O}(\frac{N}{P}(\log_2 N)^2)$ variant of the Redistribution component on Platform 2.

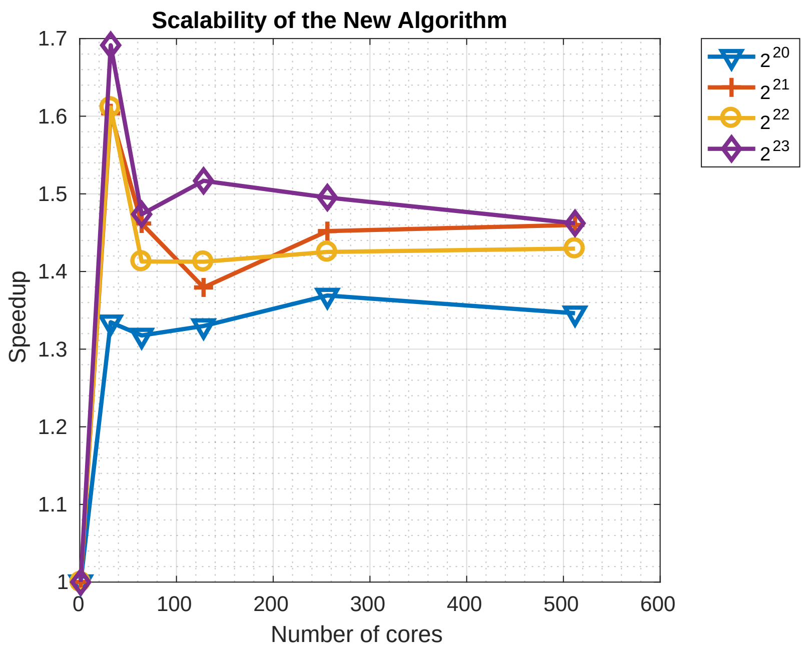


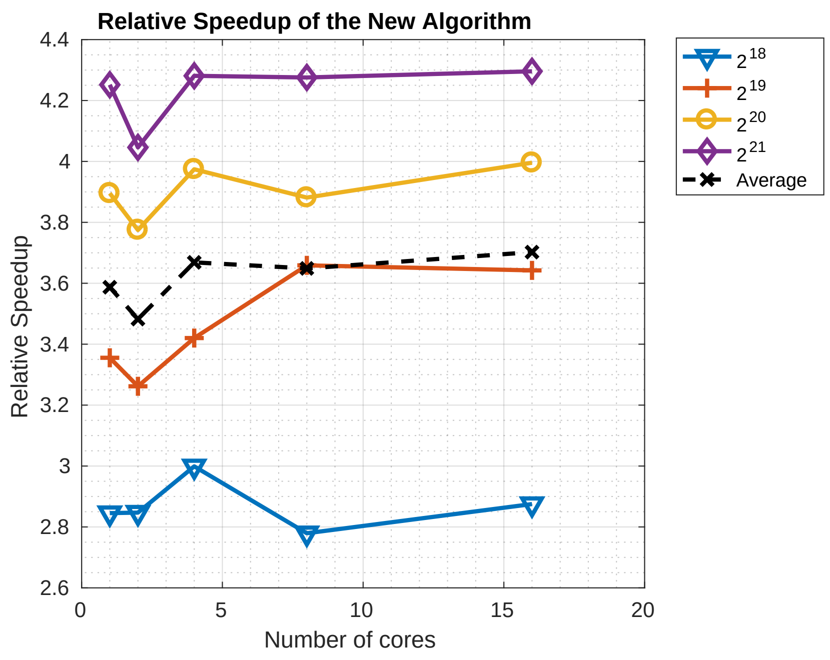
FIGURE 4.22: Scalability of the $\mathcal{O}(\frac{N}{P}(\log_2 N)^2)$ variant of the Redistribution component on Platform 2.

The relative speedup of the $\mathcal{O}(\frac{N}{P}(\log_2 N)^2)$ variant of the redistribution component (relative to the $\mathcal{O}(\frac{N}{P}(\log_2 N)^3)$ variant) is significant in all cases from a factor of 2 (Platform 1) and 24 (Platform 2). For both platforms, this speedup increases as the number of particles increases. However, with Platform 1, which has a single node such that all cores share memory, the speedup decreases as the number of cores increases for a fixed number of particles. In contrast, with Platform 2, the speedup is constant for large numbers of cores.

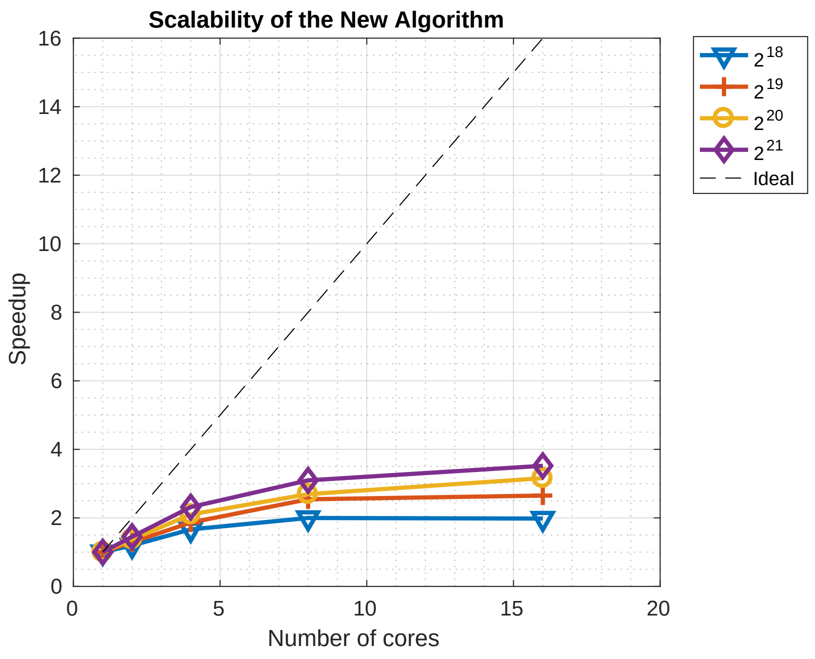
The scalability of the $\mathcal{O}(\frac{N}{P}(\log_2 N)^2)$ variant of the redistribution component is far from ideal as an increasing number of cores culminates in minimal (if any) improvements in performance. This occurs because, in the context of both Platforms, it is the communication, and not the computation, that limits performance. This observation also explains why the larger number of cores in Platform 2 does not offer improved scalability relative to Platform 1. The processors of Platform 2 are distributed across multiple nodes and communicate across a network, whereas those on Platform 1 are all part of the same node and communicate using shared memory.

4.3.5.5.2 Resulting overall particle filter performance

Figures 4.23 and 4.24 describe the speedup and scalability of the overall particle filter using the $\mathcal{O}(\frac{N}{P}(\log_2 N)^2)$ redistribution component in the context of Platforms 1 and 2, respectively.

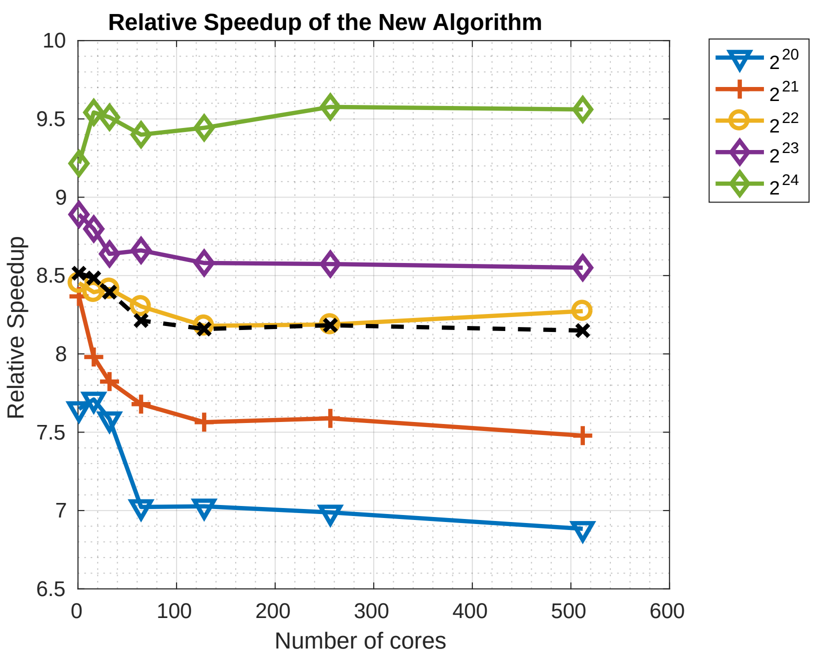


(a) Relative Speedup of $\mathcal{O}(\frac{N}{P}(\log_2 N)^2)$ variant

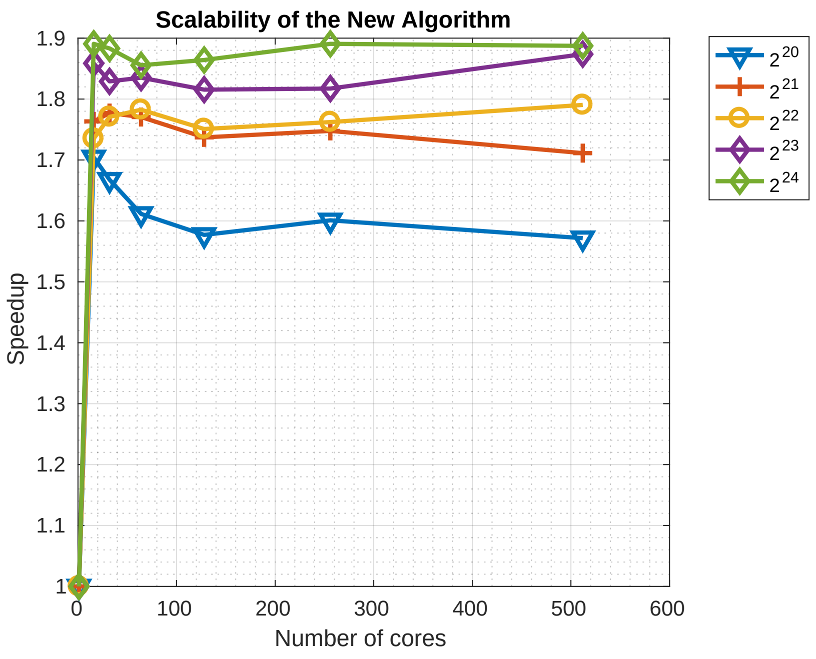


(b) Scalability of the $\mathcal{O}(\frac{N}{P}(\log_2 N)^2)$ variant

FIGURE 4.23: Relative Speedup and Scalability of the overall particle filter algorithm using the $\mathcal{O}(\frac{N}{P}(\log_2 N)^2)$ variant of the Redistribution component on Platform 1. The average is used to give some intuition based on the considered input values.



(a) Relative Speedup of $\mathcal{O}(\frac{N}{P}(\log_2 N)^2)$ variant



(b) Scalability of the $\mathcal{O}(\frac{N}{P}(\log_2 N)^2)$ variant

FIGURE 4.24: Relative Speedup and Scalability of the overall particle filter algorithm using the $\mathcal{O}(\frac{N}{P}(\log_2 N)^2)$ variant of the Redistribution component on Platform 2. The average is used to give some intuition based on the considered input values.

The speedup factors, as measured in the context of the overall particle filter algorithm, are between 3 and 9.5. Again, for both platforms, the speedup increases with the number of particles. Again, the scalability is far from ideal.

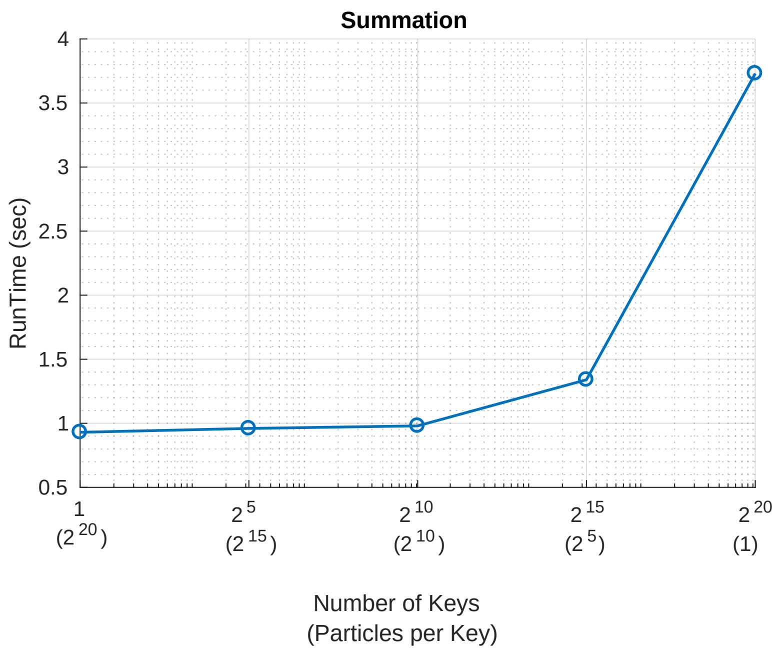


FIGURE 4.25: Performance of summation using Spark with a fixed total number of values comprised of different number of keys and therefore different numbers of values per key.

4.3.6 Discussion

The goal of this research is to reduce the execution time of particle filters dramatically. While we gained significant insights from the performance analysis described above, the results are disappointing. Using the combination of algorithms and hardware considered, we cannot improve on the execution speed achieved by a naïve redistribute implementation.

At one level, this is because the baseline against which we are comparing performance is relatively simple and mature. Our corresponding implementation is therefore relatively well optimised. In contrast, our proposed implementation is novel and has not been significantly optimised. However, we do not see it as fruitful to optimise our current implementation as we suggest two other issues cause these disappointing results.

The first issue is that, in our implementations, we assumed each particle has a unique key in the MapReduce framework. There are, therefore, as many keys as there are particles. To understand the potential benefit of having more than one value per key, we investigate how the performance of summation, in the context of a single core in Platform 1 and 2^{20} values, changes as a function of the number of values per key. Figure 4.25 highlights that, for the example of summation in the context of a specific hardware configuration, a fourfold improvement in execution speed is possible by changing the number of values per key. This implies that runtime could change significantly if other components considered multiple particles to be associated with each key.

However, the primary issue limiting runtime is the MapReduce framework. As discussed in Section 4.3.2.2, before every reduce operation, the values associated with each key are collated. This is useful in the context of applications where the number of values associated with each key and the number of unique keys is unknown (e.g., where the task is to count the number of occurrences of each word in a set of documents). However, in the particle filter application, the number of unique keys is known to be the number of particles, and the algorithms are chosen such that the number of values for each key are pre-defined for each algorithmic component. The flexibility that MapReduce provides offers no utility for the particle filter implementation, which itself is not an issue, but this flexibility is achieved through a "shuffle-and-sort" phase that precedes every reduce operation. This phase, as is self-evident from its name, is single-core bound in the versions or frameworks we utilised. This sort is demanding in terms of communication and processing. So, every time MapReduce performs even simple operations (e.g., cumulative sum), it is likely that the infrastructure is collating the keys and sorting them. Given the significant quantity of simple operations involved in our particle filter operation, we presume this overhead dominates the execution time.

This observation motivates consideration of alternative frameworks which do not provide for the same flexibility offered by MapReduce requiring such an overhead. Our research efforts, therefore, consider rethinking the implementation with alternative, lower-level frameworks.

4.3.7 Summary

In this section, we designed an improved parallel particle filtering algorithm. The core feature is a novel redistribution component providing a deterministic runtime and time-complexity of $\mathcal{O}(\frac{N}{P}(\log N)^2)$ for N particles and N processors). This improves a previous approach that achieved a time-complexity of $\mathcal{O}(\frac{N}{P}(\log N)^3)$.

A particle filter, including the previous and new redistribution components, was implemented using two Big Data frameworks, Hadoop and Spark. Instead of assuming the performance of such an implementation is faster compared to a single core version, extensive performance evaluations were conducted. Our new component outperforms the original version in isolation and when a particle filter uses the new component in place of the original. Our results indicate that, in the context of a particle filter, Spark's ability to perform calculations in memory enable it to offer a 25-fold improvement in runtime relative to Hadoop. Using Spark and our new component, we showed that, as the number of particles increases, so does the implementation efficiency.

This performance evaluation highlights it is not always valid to assume that porting algorithms to Big Data frameworks will increase execution speed. Indeed, the implementation we evaluated is limited by the communications overhead necessarily associated with giving each particle a unique key as does the MapReduce framework. As a result, while we can achieve a speedup of 3-fold with 16 cores in a single node, with 512 cores spread across 28 nodes, we only achieve a speedup of approximately 1.4. Furthermore, our implementation is outperformed by a naïve implementation by a factor of approximately 20. In other words, using our current implementation, we cannot outperform an optimised single processor resampling algorithm.

Of course, there will be applications where resampling is a small fraction of the total computational cost of the particle filter. In such contexts, the proposal, likelihood or dynamic model will be computationally demanding to calculate, while these components of the particle filter are trivial to parallelise. Our future work will broaden the applicability of our results beyond these applications. Specifically, we plan to focus on architectures involving a single key being related to multiple particles, explicitly minimising the need for data movement, and removing the large lineages that appear to be limiting the performance possible using Spark.

Finally, our implementations are available for public access via an open source repository at GitHub as particlefilter `particlefilter` [7].

4.4 Conclusions

This chapter reviewed a selected number of approaches for parallel resampling methods. A novel parallel resampling method was proposed and implemented in MapReduce. This new method improves the time complexity of previous research, and our results illustrated the benefits of the new methodology. Future work will extend the analysis and benefits of the proposed method in hardware platforms, software, and hardware optimisation and analysis using real-time applications on high-dimensional spaces.

Chapter 5

Efficient particles recycling

5.1 Introduction

The particles recycling method is a mechanism proposed in [78] as an alternative approach to making estimations on the posterior (or target) distribution in Sequential Monte Carlo (SMC) samplers. In the basic SMC sampler (as described in Chapter 3), estimations on the posterior distribution are computed using only the particles of the last iteration. The particles recycling method performs estimations using the particles from all iterations without the need to discard particles. Practically, assuming K iterations and N particles, estimations in the original algorithm are achieved using the last N generated particles, while in the particles recycling all $K \cdot N$ particles are considered.

In this chapter, a novel recycling method is proposed, applied in high dimensional static distributions, and compared with the traditional and existing estimation approaches (i.e., the method proposed in [78]) . Both recycling methods are demonstrated to outperform the traditional algorithm by leading to faster convergence, while the proposed approach is more efficient than the existing method. In Section 5.2, the traditional, existing, and new methods are discussed followed by an evaluation in Section 5.3 and conclusions in Section 5.4.

5.2 Estimation methodologies

Three methodologies for computing estimations in SMC samplers are discussed. The traditional approach, the existing method from [78], and our new proposed methodology.

5.2.1 Basic method

In the basic SMC sampler, estimations of a function of interest, $f(.)$, over the posterior distribution, $\pi(.)$, are computed using only the particles of the last iteration as

$$\bar{f} = \mathbb{E}_{\pi}\left[f\right] = \int \pi\left(x\right) f\left(x\right) dx \approx \sum_{i=1}^{N} w_k^{(i)} f\left(x_k^{(i)}\right) \tag{5.1}$$

or alternatively

$$\bar{f} = \mathbb{E}_{\pi}\left[f\right] = \int \pi\left(x\right) f\left(x\right) dx \approx \sum_{i=1}^{N} \tilde{w}_k^{(i)} f\left(x_k^{(i)}\right) \tag{5.2}$$

and

$$\underbrace{\sum_{i=1}^{N} w_k^{(i)} f\left(x_k^{(i)}\right)}_{\hat{f}} \neq \underbrace{\sum_{i=1}^{N} \tilde{w}_k^{(i)} f\left(x_k^{(i)}\right)}_{\tilde{f}} \tag{5.3}$$

where w and $\tilde{w}_k^{(i)}$ correspond to the importance weights and the normalised (i.e., the particles sum to unity) importance weights. For example, the mean value of the posterior distribution, $\pi(.)$, is approximated by multiplication of the particles during the last iteration, x_k with the corresponding normalised weights, $\tilde{w}_k$. As explained in the Appendix B.1, the $\hat{f}$ is unbiased estimator of the posterior expectation, $\bar{f}$, of any function f. The expectation is approximated using the normalised weights, $\tilde{f}$, which is a biased estimator of $\hat{f}$.

5.2.2 Existing method

The existing method makes estimations of the posterior distribution using the particles generated during all the iterations of the SMC sampler. In every iteration, estimations of the intermediate distributions are computed using Equation 5.1. The final estimation, referring to the posterior distribution, is computed using all the intermediate estimations. The intermediate distributions do not directly target the posterior, $\pi(.)$. To correct this issue, importance sampling identity is applied to each of the samples generated during every iteration of the SMC sampler [78], so the estimates are determined by

$$\hat{f} = \frac{\sum_{j=1}^{K} \hat{f}_j \tilde{c}_j}{\sum_{j=1}^{K} \tilde{c}_j} = \frac{\sum_{j=2}^{K} \pi(x_j^{(i)}) \tilde{w}_j^{(i)} \tilde{w}_{j-1}^{(i)}}{\sum_{j=2}^{K} \tilde{w}_{j-1}^{(i)}} \tag{5.4}$$

where i denotes the particle index and c_j, as discussed in Appendix B.1, is the normalising constant of the joint density

$$\tilde{c}_j \approx c_j = \int \pi(x_j)dx_j \tag{5.5}$$

5.2.3 New method

Algorithm 13 Proposed Particles Recycling SMC Sampler

```
1: for $i = 1 : N$ do
2:     Sample $x_1^{(i)} \sim q(x_1)$
3:     Calculate $w_1^{(i)} = \frac{\pi(x_1^{(i)})}{q(x_1^{(i)})}$
4: end for
5: for $k = 2 : K$ do
6:     for $i = 1 : N$ do
7:         Sample $x_k^{(i)} \sim q(x_k^{(i)}|x_{k-1}^{(i)})$
8:         Calculate $w_k^{(i)} = w_{k-1}^{(i)} \frac{\pi(x_k^{(i)})L(x_{k-1}^{(i)}|x_k^{(i)})}{\pi(x_{k-1}^{(i)})q(x_k^{(i)}|x_{k-1}^{(i)})}$
9:     end for
10:    Normalising constant, $\tilde{c}_k = \frac{w_{k-1}\pi(x_k)}{q(x_k|x_{k-1})}$
11:    Weights Normalisation $\tilde{w}_k = \frac{w_k}{\sum(w_k)}$
12:    Calculate the effective sample size $N_{eff} = \frac{1}{\sum_{i=1}^{N}(\tilde{w}_k^{(i)})^2}$
13:    if $N_{eff} < N_T$ then
14:        Resampling with $\tilde{w}_k$, and produce new particles population, $x_k$
15:        Set $w_k = \frac{1}{N}$
16:    end if
17: end for
18: Particles recycling calculation using the equation 5.10
```

The difference between the proposed and existing methodologies is the computation of the normalising constant of the joint density (i.e., Equation 5.5), which is computed [78] for every iteration of the SMC sampler as $\tilde{c}_k = \sum_{j=2}^{K} \tilde{w}_{j-1}^{(i)}$, where $i = 1, 2, \ldots, N$ denotes the particle index with N total number of particles. For the proposed particles recycling method, the normalising constant is $\tilde{c}_k = \sum_{j=2}^{K} \frac{\pi(x_j^{(i)})}{q(x_j^{(i)}|x_{j-1}^{(i)})} \tilde{w}_{j-1}^{(i)}$. More precisely, in the new method the Equation 5.5 is computed as

$$c_k = \int \pi\left(x_k\right) dx_k = \int \pi\left(x_k\right) \underbrace{\sum_{i=1}^{N} \tilde{w}_{k-1}^{(i)}}_{=1} dx_k \tag{5.6}$$

$$= \sum_{i=1}^{N} \int \pi\left(x_k^{(i)}\right) \tilde{w}_{k-1}^{(i)} \underbrace{\frac{q\left(x_k^{(i)}|x_{k-1}^{(i)}\right)}{q\left(x_k|x_{k-1}^{i}\right)}}_{=1} dx_k \tag{5.7}$$

$$\approx \tilde{c}_k = \sum_{i=1}^{N} \frac{1}{N'} \sum_{j=1}^{N'} \frac{\pi\left(x_k^{(j)}\right)}{q\left(x_k^{(j)}|x_{k-1}^{(i)}\right)} \tilde{w}_{k-1}^{(i)} \tag{5.8}$$

$$\text{Consider } i = j \text{ and } N' = 1 \implies \tilde{c}_k = \sum_{i=1}^{N} \frac{\pi\left(x_k^{(i)}\right)}{q\left(x_k^{(i)}|x_{k-1}^{(i)}\right)} \tilde{w}_{k-1}^{(i)} \tag{5.9}$$

The N' denotes multiple samples per the sample of the previous iteration. In Equation B.10 it is assumed that $N' = 1$. The new algorithm estimations are computed as

$$\hat{f} = \frac{\sum_{j=1}^{K} \hat{f}_j \tilde{c}_j}{\sum_{j=1}^{K} \tilde{c}_j} = \frac{\sum_{j=2}^{K} \pi(x_j^{(i)}) \tilde{w}_j^{(i)} \frac{\pi(x_j^{(i)}) \tilde{w}_{j-1}^{(i)}}{q(x_j^{(i)}|x_{j-1}^{(i)})}}{\sum_{j=2}^{K} \frac{\pi(x_j^{(i)}) \tilde{w}_{j-1}^{(i)}}{q(x_j^{(i)}|x_{j-1}^{(i)})}} \tag{5.10}$$

where the $i = \{1, 2, \ldots N\}$ denotes the number of particles. The pseudocode of the proposed particles recycling SMC sampler is available in Algorithm 13 and can be compared with the traditional SMC sampler in Algorithm 5. The particles recycling method does not influence the overall time complexity of the algorithm, which is $\mathcal{O}(\frac{P}{N}(\log N)^2)$ for the SMC sampler and particle filter according to [97]. Also, the time complexity of the particles recycling method is equivalent to the time complexity of the sum, max, and min algorithmic components, which is $\mathcal{O}(\frac{P}{N}(\log N))$. Thus, the particles recycling is a computationally trivial mechanism.

5.3 Simulations

In the following section, the original SMC sampler and the SMC sampler with the two recycling methods are compared using high dimensional static distributions. The goal is to generate samples from the static distributions and estimate the true mean value. The results highlight the importance of the particle recycling mechanism and the efficiency improvement of the new proposed particle recycling methodology over the traditional and existing methods [78].

The proposal distribution is a random walk, and the backward kernel is selected to emulate MCMC, $L(x_{k-1}|x_k) = q(x_k|x_{k-1})$. The estimations are computed using

$N = 100$ particles and $K = 100$ iterations or $N \cdot K = 10000$ samples of the target distribution. While different configurations for the number of particles and iterations could be examined, they are not considered in the evaluation. Every experiment corresponds to an average of 100 Monte Carlo runs. In Section 5.3.1, the target (or posterior) distribution is a zero mean N-dimensional Gaussian distribution with covariance of the identity matrix.

$$\pi(x) = \frac{1}{(2\pi)^{\frac{d}{2}} |\Sigma|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)\right)$$

In Section 5.3.2, the target distribution is a multivariate Student's t distribution with seven degrees of freedom.

$$\pi(x) = -log(\Gamma(\frac{\nu+d}{2})) - log(\Gamma(\frac{\nu}{2})) + (-\frac{(\nu+d)}{2} log(1+\frac{\|x\|^2}{\nu})) - \frac{d}{2} log(\nu\pi) \tag{5.11}$$

In Section 5.3.3, the target distribution is the Ackley function.

$$\pi(x) = 20exp(-0.2\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}) + exp(\frac{1}{d}\sum_{i=1}^{d} cos(2\pi x_i)) - 20 - exp(1)$$

where d corresponds to the input size. In Figure 5.3 the different methods are compared by increasing the number of particles. In Figures 5.2, 5.3, 5.4, 5.6 (or Tables 5.1, 5.2, 5.3 and 5.4, respectively) we denote with:

1. Method $\mathbf{m_1}$: The basic estimation.
2. Method $\mathbf{m_2}$: The new proposed particle recycling algorithm.
3. Method $\mathbf{m_3}$: The existing particle recycling algorithm proposed in [78].

The basic SMC sampler perform estimations using only the last 100 particles or only the 1% of the total number of samples. When we consider the particle recycling methods, all samples are used for the final estimation. Both recycling methods are significantly better than the basic SMC sampler. This improvement appears to further increase as the dimensionality of the target distribution increases. For example, this effect is noticeable in both the Gaussian and Student's t distributions. The different methods are applied in a more complicated function (e.g., the Ackley function), which has a large number of local maximum and a single global maximum, and is commonly used in testing optimisation algorithms. The SMC sampler, in this case, appears to struggle to converge, especially when the dimensionality increases. The new proposed method outperforms the other two estimation methodologies. There is a potential the existing method to outperform the new methodology (e.g. Table 5.1 and Figure 5.2) in high dimensional posterior distributions. An explanation for this behaviour is potentially related with the assumptions considered in Equation 5.9. However, it is illustrated in Table 5.2 and Figure 5.2 that the new method can achieve better accuracy by considering more particles and iterations.

5.3.1 N-dimensional Gaussian distribution

TABLE 5.1: Comparison of the recycling methods on the Gaussian distribution using 10000 samples (100 particles) based on the log mean squared error (estimation of the mean value).

Dimensions	**Basic** (m_1)	**Proposed** (m_2)	**Existing** (m_3)
2	-3.56	-5.64	-4.87
4	-2.63	-5.10	-4.72
6	-2.21	-4.80	-4.54
8	-1.85	-4.47	-4.36
10	-1.64	-4.26	-4.26
12	-1.50	-4.01	-4.10

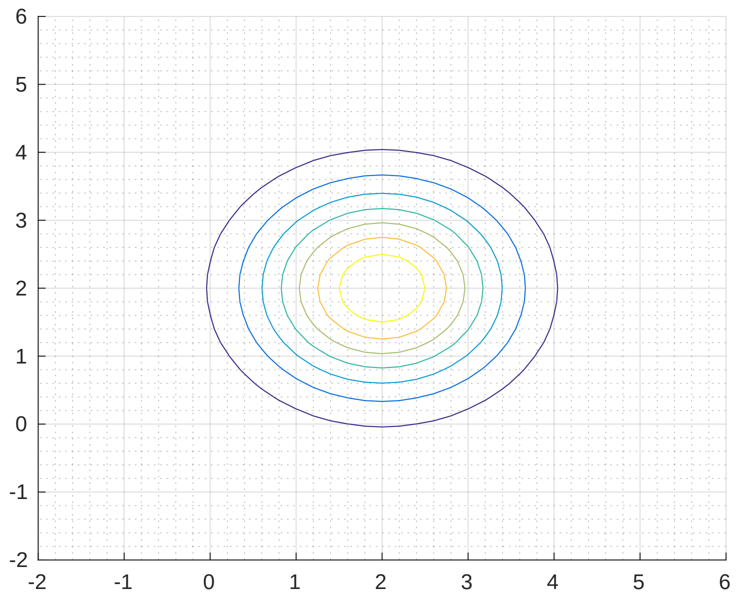


FIGURE 5.1: Exemplar of a multivariate Gaussian distribution (estimation of the mean value)

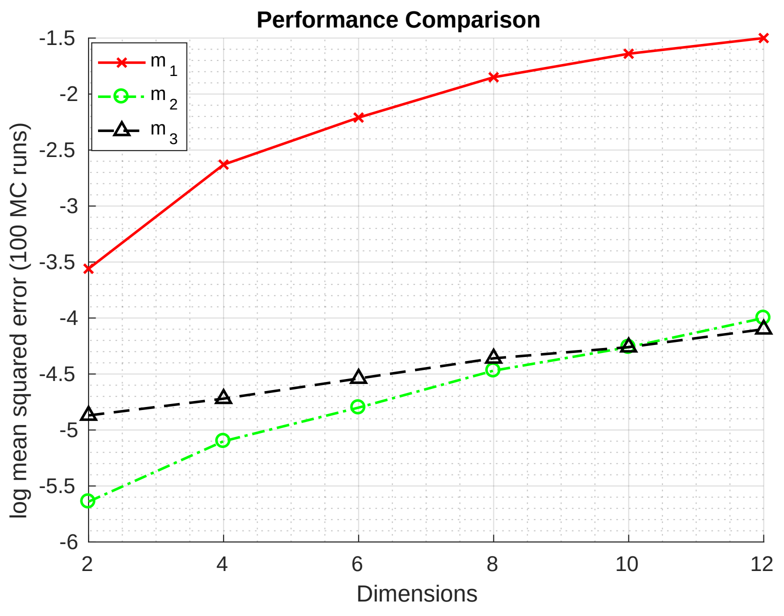


FIGURE 5.2: Comparison using 10000 samples (100 particles). Figure 5.1 illustrates the target distribution.

TABLE 5.2: Comparison on a 10-dimensional Gaussian distribution through increasing the number of samples (estimation of the mean value).

Samples	**Basic** (m_1)	**Proposed** (m_2)	**Existing** (m_3)
20000	-1.91	-4.51	-4.41
30000	-1.99	-4.55	-4.35
40000	-2.23	-4.57	-4.41
50000	-2.36	-4.79	-4.57

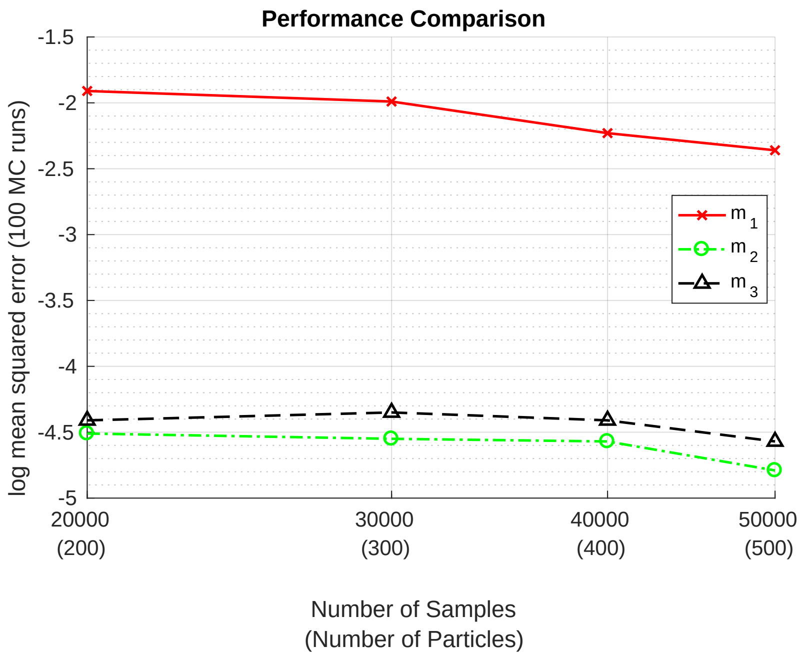
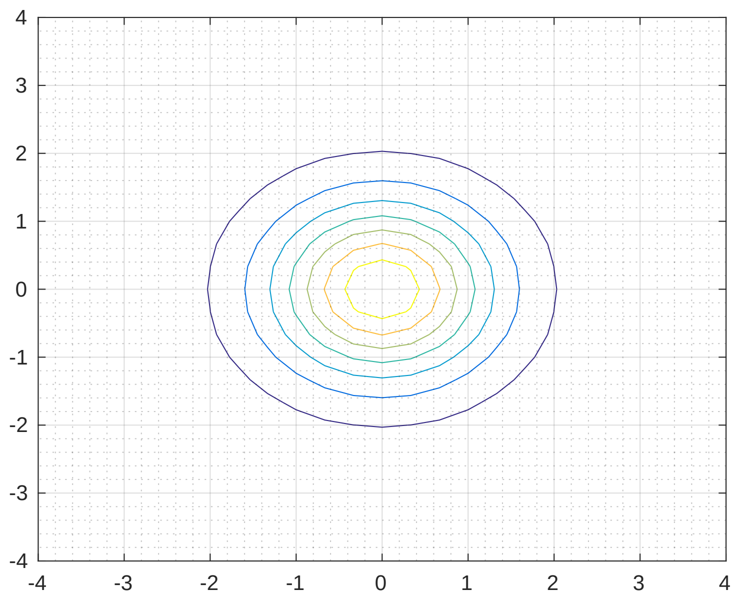


FIGURE 5.3: Comparison on a 10-dimensional Gaussian distribution through increasing the number of samples (estimation of the mean value).

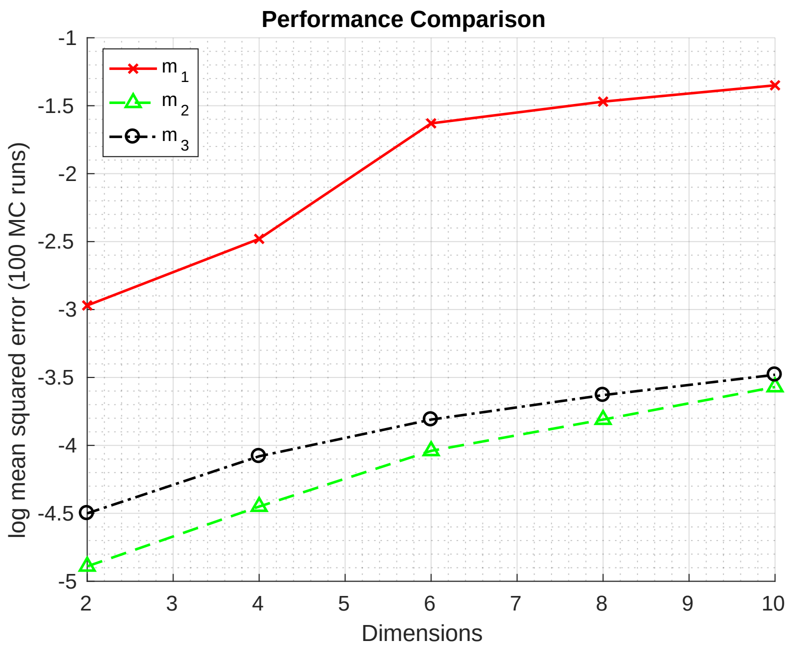
5.3.2 N-dimensional Student's t distribution

TABLE 5.3: Comparison of the recycling methods on the Student's t distribution (estimation of the mean value) using 10000 samples (100 particles).

Dimensions	**Basic** (m_1)	**Proposed** (m_2)	**Existing** (m_3)
2	-2.97	-4.89	-4.50
4	-2.48	-4.45	-4.08
6	-1.63	-4.04	-3.81
8	-1.47	-3.81	-3.63
10	-1.35	-3.57	-3.48



(a) Exemplar of the target distribution



(b) Performance Comparison

FIGURE 5.4: (a) Exemplar of a multivariate Student's t distribution (estimation of the mean value), and (b) comparison with 10000 samples (100 particles).

5.3.3 N-dimensional Ackley function

TABLE 5.4: Comparison of the recycling methods on the Ackley function (estimation of the mean value) using 10000 samples (100 particles).

Dimensions	**Basic** (m_1)	**Proposed** (m_2)	**Existing** (m_3)
2	-3.92	-6.19	-5.43
4	-1.71	-3.27	-2.98
6	-0.10	-1.31	-1.23
8	1.00	-0.16	-0.05
10	1.66	0.22	0.49

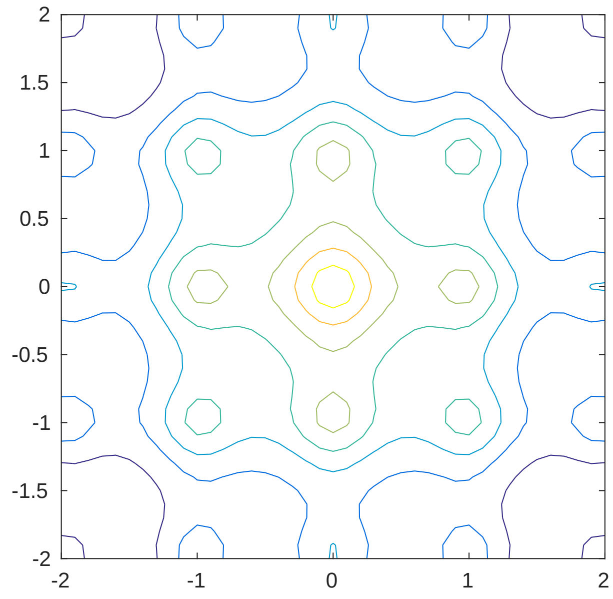
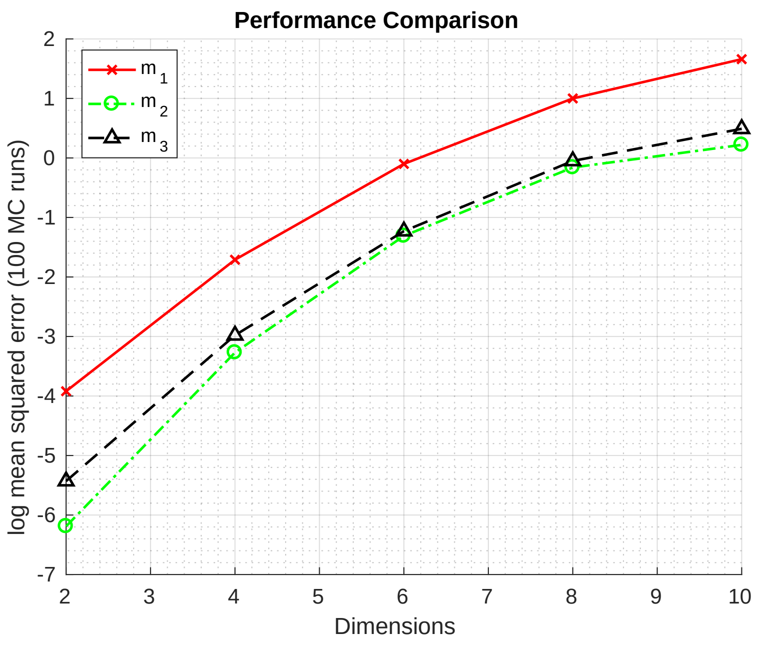


FIGURE 5.5: Exemplar of the multivariate inverse Ackley function (estimation of the mean value).



(b) Performance Comparison

Figure 5.6: Comparison using 10000 samples (100 particles). Figure 5.5 illustrates the target distribution.

5.4 Conclusions

A novel particle recycling strategy was proposed and compared with the traditional and existing estimation methodologies for SMC samplers. The recycling methodologies are more accurate than the basic SMC sampler where estimations are computed using the particles during the last iterations only. The particles recycling is a computationally trivial mechanism, where the time complexity is equivalent with the sum, max, and min algorithmic methods of $\mathcal{O}(\frac{P}{N}(\log N))$. The potential of the proposed methodology is demonstrated with sampling from high dimensional static distributions.

Chapter 6

Selecting the forward Markov kernel

6.1 Introduction

In the Metropolis algorithm and the Sequential Monte Carlo methods, the proposal distribution is a user-defined probability density function. The traditional approach uses a random walk forward transition kernel. The applicability of this approach is relatively easy for both methodologies. In the Metropolis, algorithm the random walk proposal suffers from a large number of rejected samples as the new proposed samples are independent of the target distribution [31]. A high acceptance rate in the random walk proposal implies that the convergence is very slow. This observation suggests a path of exploration for the choice of the proposal distribution. Several publications are devoted on considering alternative proposal distributions that are more efficient than the random walk. These proposal distributions include the Ozaki and Euler discretisation in [27] and [31], the partially implicit local linearisation in [22], and the Ornstein-Uhlenbeck process and semi-implicit Euler discretisation in [32], which all focus on proposals with better convergence characteristics to improve the efficiency of the Metropolis algorithm by reaching stationarity faster.

In this chapter, the Euler discretisation and the partically implicit local linearisation are applied and compared to the random walk proposal. The results demonstrate that considering more sophisticated proposal distributions compared to the random walk can improve the accuracy of the SMC sampler.

Section 6.2 discusses the forward kernel proposal using the Euler discretisation. In Section 6.3, three high dimensional experiments show the benefits of considering more efficient forward Markov kernels compared to the random walk proposal. Final thoughts and future work are reviewed in Section 6.4.

6.2 Langevin diffusion

6.2.1 Fokker-Plank equation

In the one-dimensional case, diffusion is a stochastic process which can be written as a stochastic differential equation, based on Ito's representation [50], such that

$$dx_t = \mu\left(x_t\right)dt + \sqrt{\sigma\left(x_t\right)}dw_t \tag{6.1}$$

where x_t is a diffusion process with time index t, $\mu\left(x_t\right)$ is the drift term, $\sigma\left(x_t\right)$ the volatility and w_t is the Wiener or Brownian process. The drift term defines the mean velocity, the volatility is the covariance of the process, and the Wiener or Brownian process determines the noise (i.e., randomness). A diffusion process is a set of random variables where each is indexed with the time t. Equation 6.1 enables us to calculate the transition probability density, which we can use to calculate the expectation value of observables of a diffusion process [81].

The Fokker-Planck equation, also known as the Kolmogorov forward equation, is a partial differential equation describing the time evolution of the probability density function, $\pi\left(x_t\right)$. The Langevin from Equation 6.1 is reformulated [53] into the following Fokker-Planck equation as

$$\frac{\partial\pi\left(x_t\right)}{\partial t} = \frac{\partial}{\partial x_t}\left(\mu\left(x_t\right)\pi\left(x_t\right)\right) + \frac{1}{2}\frac{\partial^2}{\partial {x_t}^2}\left(\sigma\left(x_t\right)\pi\left(x_t\right)\right) \tag{6.2}$$

The simplest form of a diffusion process is the standard Brownian, which is generated by the stochastic differential equation

$$dx_t = dw_t \tag{6.3}$$

where the drift term is zero, and the volatility is one. In this case, the time evolution of the probability density function, $\pi\left(x_t\right)$, satisfies the Fokker-Planck equation simplifying to

$$\frac{\partial\pi\left(x_t\right)}{\partial t} = \frac{1}{2}\frac{\partial^2}{\partial {x_t}^2}\left(\pi\left(x_t\right)\right) \tag{6.4}$$

The Fokker-Planck equation shows the statistical behaviour of dynamical systems and is used to solve Langevin equations. The non-linear Langevin equations are not easy to solve, and its reformulation using Fokker-Planck provides a computationally approachable solution [111].

6.2.2 Discrete time Langevin diffusion

Assume that the volatility in equation 6.1 is constant:

$$\sigma\left(x_t\right) = \sigma \tag{6.5}$$

and that the drift is defined as follows:

$$\mu\left(x_t\right) = -\frac{1}{2}\sigma\frac{\partial}{\partial x_t}\log\pi\left(x_t\right) \tag{6.6}$$

For any $f\left(x\right)$:

$$\frac{\partial}{\partial x}\log f\left(x\right) = \frac{1}{f\left(x\right)}\frac{\partial f\left(x\right)}{\partial x} \tag{6.7}$$

such that

$$\mu\left(x_t\right) = -\frac{1}{2}\sigma\frac{1}{\pi\left(x_t\right)}\frac{\partial\pi\left(x_t\right)}{\partial x_t} \tag{6.8}$$

Substituting Equations 6.8 and 6.5 into 6.2, we obtain

$$\begin{aligned}\frac{\partial\pi\left(x_t\right)}{\partial t} =& \frac{\partial}{\partial x_t}\left(-\frac{1}{2}\sigma\frac{1}{\pi\left(x\right)}\frac{\partial\pi\left(x\right)}{\partial x_t}\pi\left(x_t\right)\right) + \frac{1}{2}\frac{\partial^2}{\partial {x_t}^2}\left(\sigma\pi\left(x_t\right)\right) && (6.9)\\ =& \frac{\sigma}{2}\left[-\frac{\partial}{\partial x_t}\left(\frac{\partial\pi\left(x_t\right)}{\partial x_t}\right) + \frac{\partial^2\pi\left(x_t\right)}{\partial {x_t}^2}\right] && (6.10)\\ =& \frac{\sigma}{2}\left[-\frac{\partial^2\pi\left(x_t\right)}{\partial {x_t}^2} + \frac{\partial^2\pi\left(x_t\right)}{\partial {x_t}^2}\right] && (6.11)\\ =& 0 && (6.12)\end{aligned}$$

such that if Equations 6.8 and 6.5 are true, then simulating from (6.1) means that

$$\pi\left(x_t\right) = \pi\left(x\right) \tag{6.13}$$

such that the samples will always be samples that are from $\pi\left(x\right)$. We can integrate Equation (6.1) over time to deduce

$$p\left(x_{t+\epsilon}|x_t\right) \approx \mathcal{N}\left(x_{t+\epsilon}; x_t + \epsilon\frac{1}{2}\sigma\frac{\partial\log\pi\left(x_t\right)}{\partial x_t}, \epsilon\sigma\right) \tag{6.14}$$

where $\mathcal{N}\left(x;\mu,\Sigma\right)$ is a Gaussian density for x parameterised by a mean of μ and a variance of Σ. When ϵ is small, (6.14) provides a high fidelity approximation. However, as ϵ increases, the approximation fidelity reduces, although it is possible to consider improved approximation schemes for what is described in Equation 6.14. For example, [27] proposes to use Ozaki's approximation method [79]) to define the Langevin Monte Carlo with Ozaki discretisation (LMCO) algorithm as an alternative to MALA. In [22] applying an implicit method for the Langevin diffusion is proposed as well as a new method (see Equation 18) as an alternative to the random walk and Euler discretisation. In our simulations, this method is considered with the proposal, similar to Equation 6.14. of

$$p\left(x_{t+\epsilon}|x_t\right) \approx \mathcal{N}\bigg(x_{t+\epsilon}; x_t + \big(I - \frac{1}{2}\frac{\partial^2 \log \pi(x_t)}{\partial x_t^2}\lambda\epsilon\big)^{-1}\big(\frac{1}{2}\frac{\partial \log \pi(x_t)}{\partial x_t}\epsilon\big), \\ \epsilon\big(I - \frac{1}{2}\frac{\partial^2 \log \pi(x_t)}{\partial x_t^2}\lambda\epsilon\big)^{-2}\bigg) \tag{6.15}$$

where the $0 \leq \lambda \leq 1$ is the implicit parameter. The special cases $\lambda = 0$, $\lambda = 0.5$ and $\lambda = 1$ describe the Euler discretisation, the stochastic generalisation of the trapezoidal method, and the backward Euler method, respectively [22].

6.3 Simulations

This section demonstrated how the user-defined forward kernel influences the accuracy of the estimation in the context of the SMC sampler as explained through sampling from three static distributions. The first scenario examines the performance of the three proposals (random walk, Euler discretisation, and partially implicit local linearisation) on three one dimensional static distributions. The remaining scenarios compare the random walk and the Euler discretisation on high dimensional static distributions. Similar results to what will be seen here have been demonstrated in the Metropolis-adjusted Langevin algorithm (MALA) (e.g., [32]).

In all the simulations there are $K = 100$ iterations and $N = 100$ particles or in total the algorithm generates 10000 samples from the posterior distribution. It is worth mentioning that computationally both proposal distributions are computationally equivalent. Computationally, both proposal distributions are computationally equivalent and could be classified as element-wise operations implying time and space computational complexity of $\mathcal{O}(1)$ and $\mathcal{O}(N)$, respectively. These complexities are different from the convergence complexities. In the MH algorithm, the convergence complexity with a random walk proposal is $\mathcal{O}(d)$, and with a proposal based on the Euler discretisation, the complexity improves to $\mathcal{O}(d^{1/3})$, where d defines the dimensionality of the posterior distribution [87].

The choice of the forward kernel opens the door for further exploration in the context of the SMC samplers. In Section 6.3.1 where the three proposals are compared on three one dimensional static distributions, the Euler discretisation outperforms all methods. While expected to outperform the random walk, it was not initially considered to perform better than the partially implicit local linearisation method [22]. An explanation for this behaviour is that all methods describe the same distribution (i.e., Gaussian) with a different approach.

In Section 6.3.2, the target (or posterior) distribution is a zero mean Gaussian distribution with covariance of the identity matrix, such that

$$\pi(x) = \frac{1}{(2\pi)^{\frac{d}{2}} |\Sigma|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)\right) \tag{6.16}$$

where μ is the mean value and Σ the covariance matrix. The computation of the first derivative is required for the new position of each particle in the Euler discretisation with

$$\pi'(x) = \frac{\partial \pi(x)}{\partial x} = -\frac{1}{(2\pi)^{\frac{d}{2}} |\Sigma|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)\right) \Sigma^{-1} (x-\mu) \tag{6.17}$$

In Section 6.3.3, the target distribution is a multivariate Student's t distribution, such that

$$\pi(x) = -log(\Gamma(\frac{\nu+d}{2})) - log(\Gamma(\frac{\nu}{2})) + (-\frac{(\nu+d)}{2} log(1+\frac{\|x\|^2}{\nu})) - \frac{d}{2} log(\nu\pi) \tag{6.18}$$

The first derivative of the target distribution is

$$\pi'(x) = \frac{\partial \pi(x)}{\partial x} = -\frac{(\nu+d)}{2}\frac{1}{\nu}\frac{1}{1+\frac{\|x\|^2}{\nu}}\frac{2|x|\,\mathrm{sgn}\,(x)}{\nu} = -\frac{\nu+d}{\nu^2} \cdot \frac{|x|\,\mathrm{sgn}\,(x)}{1+\frac{\|x\|^2}{\nu}} \tag{6.19}$$

where $\Gamma(.)$ is the Gamma distribution, $\|.\|$ is the Euclidean norm, $|.|$ is the absolute value, $\mathrm{sgn}\,(.)$ is the sign or signum function and $\delta(.)$ is the Dirac delta function. The parameter $\nu = 20$ defines the degrees of freedom, and the preconditioning matrix is the identity, which simplifies the posterior. In Section 6.3.4, the target distribution is the N-dimensional Laplace distribution, which is also known as the double exponential distribution.

$$\pi(x) = -\|\frac{x-\mu}{\Sigma}\| - 2\|\Sigma\| \tag{6.20}$$

and the first derivative of the posterior distribution:

$$\pi'(x) = \frac{\partial \pi(x)}{\partial x} = \frac{\mathrm{sgn}\,(\mu - x)}{\|\Sigma\|} \tag{6.21}$$

where μ is the mean value and the Σ the covariance. In Table 6.1, the first derivative of some of these functions is provided.

The comparisons in Figures 6.4, 6.5, 6.6 and 6.8, with the corresponding Tables 6.2, 6.3, 6.4 and 6.5, respectively, reveal the benefits of the Euler discretisation over the random walk using the new recycling methodology. The accuracy improvement is relatively significant, and the Euler discretisation outperforms the random walk proposal.

TABLE 6.1: Function Names with the Corresponding First Derivative

Function Name	**Function,** $g(x)$	**First Derivative,** $\frac{\partial g(x)}{\partial x}$
Absolute value	$\|x\|$	$\operatorname{sgn}(x)$
Signum function	$\operatorname{sgn}(x)$	$2\delta(x)$
Euclidean Norm	$\\|x\\|$	$\frac{\|x\| \operatorname{sgn}(x)}{\\|x\\|}$

6.3.1 One dimensional static distributions

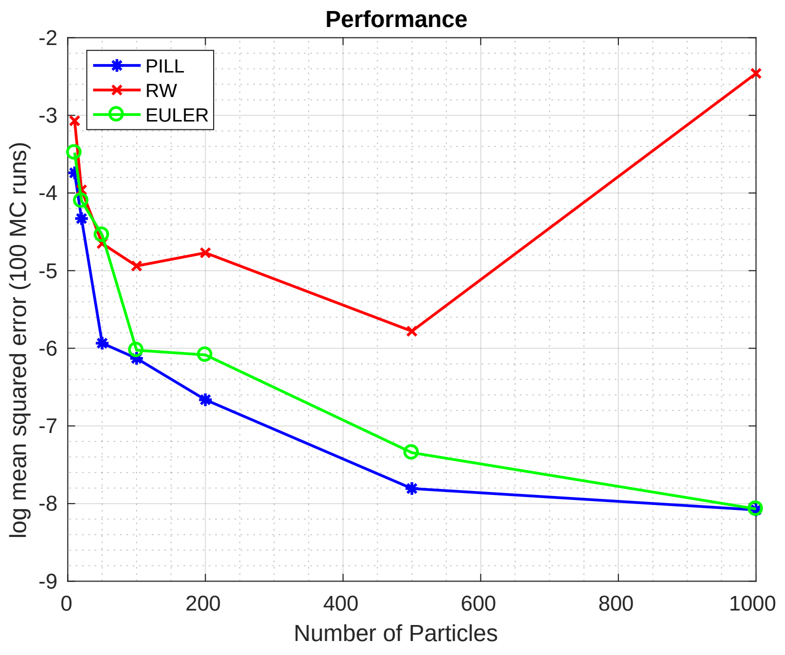


FIGURE 6.1: Comparison of the random walk, Euler discretisation, and partially implicit local linearisation on the Gaussian static distribution. In all cases the mean value of the posterior is estimated.

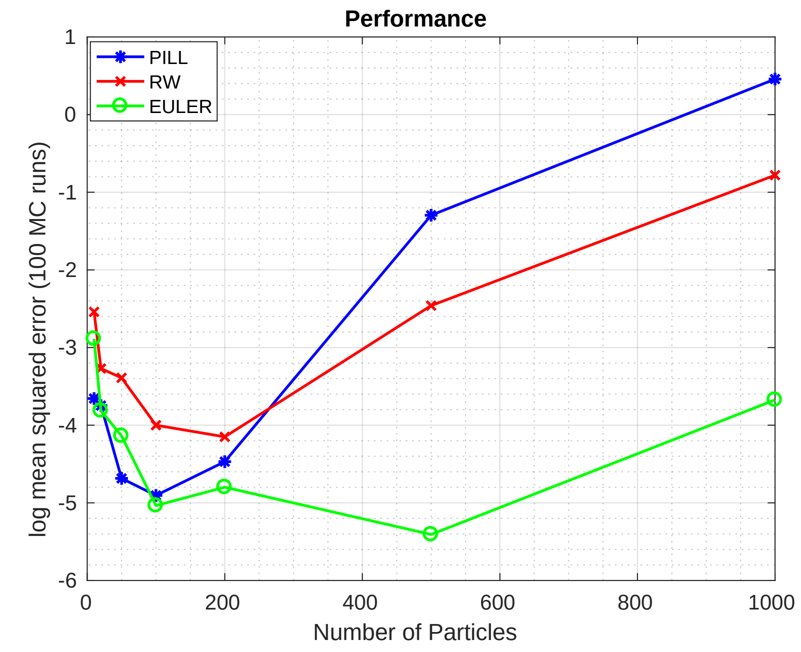


FIGURE 6.2: Comparison of the random walk, Euler discretisation, and partially implicit local linearisation on the Student's t static distribution. In all cases the mean value of the posterior is estimated.

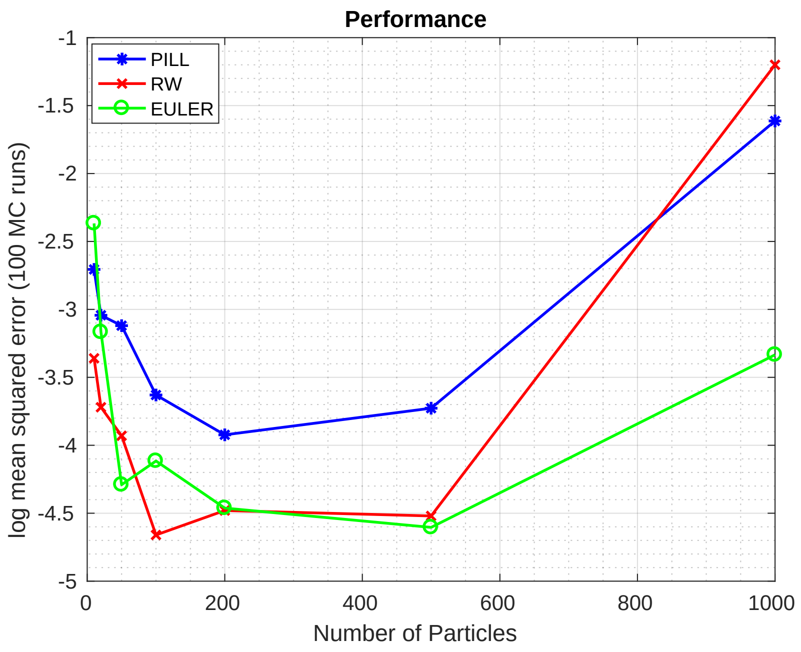


FIGURE 6.3: Comparison of the random walk, Euler discretisation, and partially implicit local linearisation on the Laplace static distribution. In all cases the mean value of the posterior is estimated.

6.3.2 N-dimensional Gaussian distribution

TABLE 6.2: Comparison of the random walk and Euler discretisation without recycling based on the log mean squared error (estimation of the mean value).

Dimensions	**Random Walk**	**Euler**
1	-4.83	-5.83
2	-3.44	-5.2
4	-2.65	-4.49
6	-2.27	-4.07
8	-2.00	-3.33
10	-1.6	-3.00

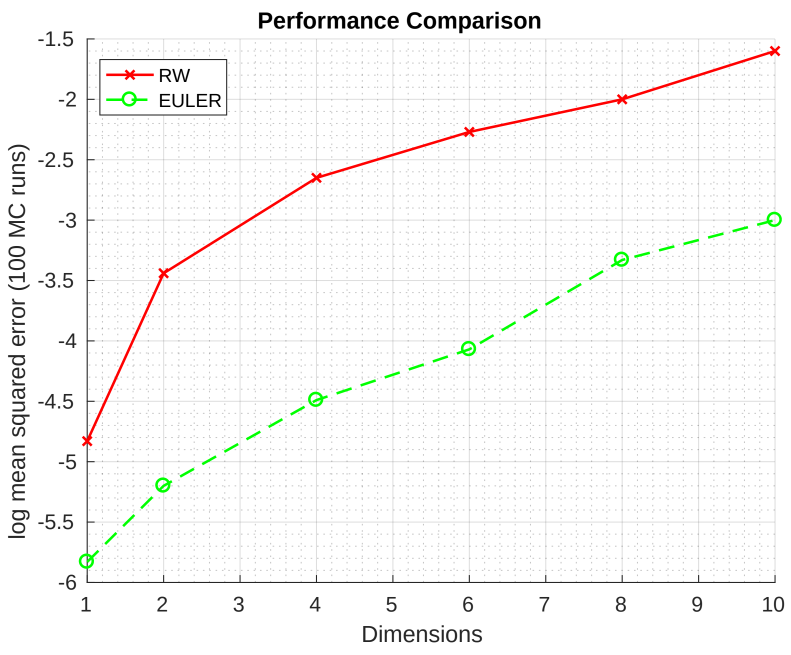


FIGURE 6.4: Comparison of the random walk and Euler discretisation without recycling based on the log mean squared error (estimation of the mean value)

Table 6.3: Comparison of the different recycling algorithms using the Euler discretisation. The m_1, m_2 and m_3 denote estimations based on the basic method, the new proposed recycling method in Chapter 5 and the method proposed in [78], respectively. In all cases the comparison is based on the log mean squared error (estimation of the mean value)

Dimensions	**Basic** (m_1)	**Proposed** (m_2)	**Existing** (m_3)
1	-5.83	-10.18	-10.01
2	-5.20	-9.00	-8.78
4	-4.49	-8.61	-8.51
6	-4.07	-8.35	-8.17
8	-3.33	-7.66	-7.55
10	-3.00	-7.40	-7.37

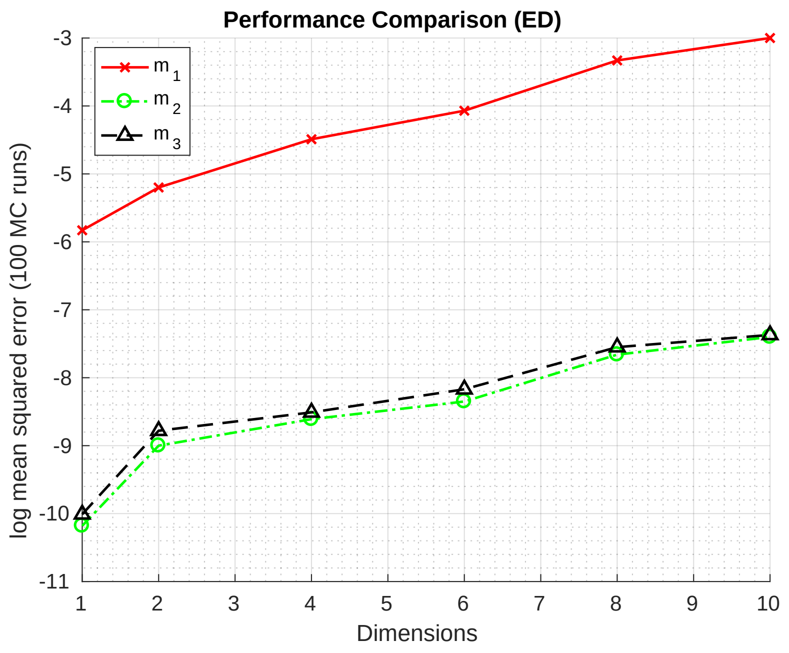
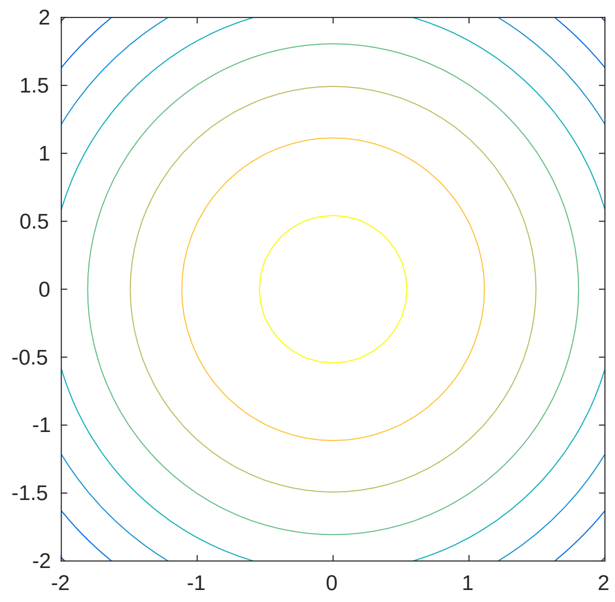


Figure 6.5: Comparison of the different recycling algorithms using the Euler discretisation based on the log mean squared error (estimation of the mean value).

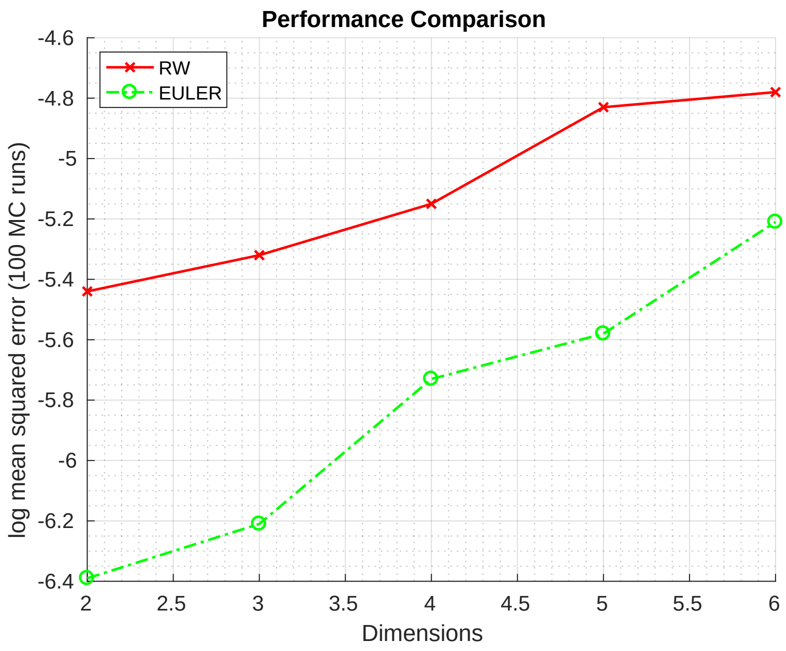
6.3.3 N-dimensional Student's t distribution

Table 6.4: Comparison of the random walk and the Euler discretisation without recycling based on the log mean squared error (estimation of the mean value).

Dimensions	**Random Walk**	**Euler**
2	-5.44	-6.39
3	-5.32	-6.21
4	-5.15	-5.73
5	-4.83	-5.58
6	-4.78	-5.21



(a) Posterior distribution



(b) Performance comparison

FIGURE 6.6: (a) Target distribution and (b) comparison of the random walk and the Euler discretisation based on the log mean squared error (estimation of the mean value).

6.3.4 N-dimensional Laplace distribution

Table 6.5: Comparison of the random walk and the Euler discretisation without recycling based on the log mean squared error (estimation of the mean value).

Dimensions	**Random Walk**	**Euler**
1	-3.81	-5.65
2	-2.24	-4.05
4	-1.29	-2.88
6	-0.65	-2.46

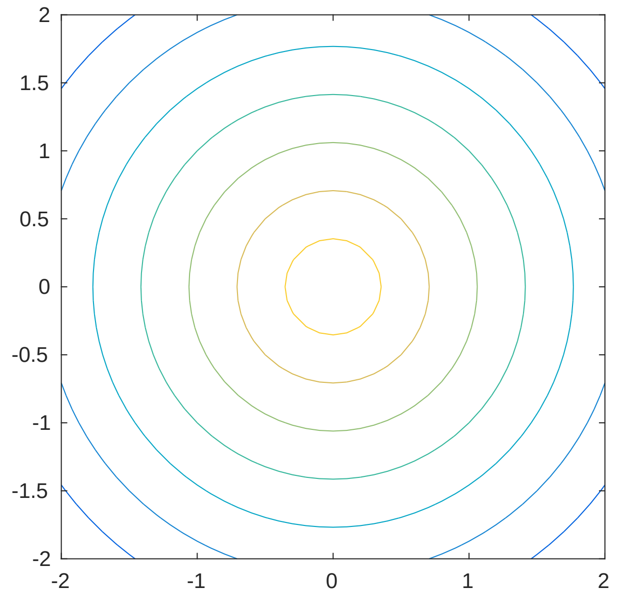


Figure 6.7: Target distribution.

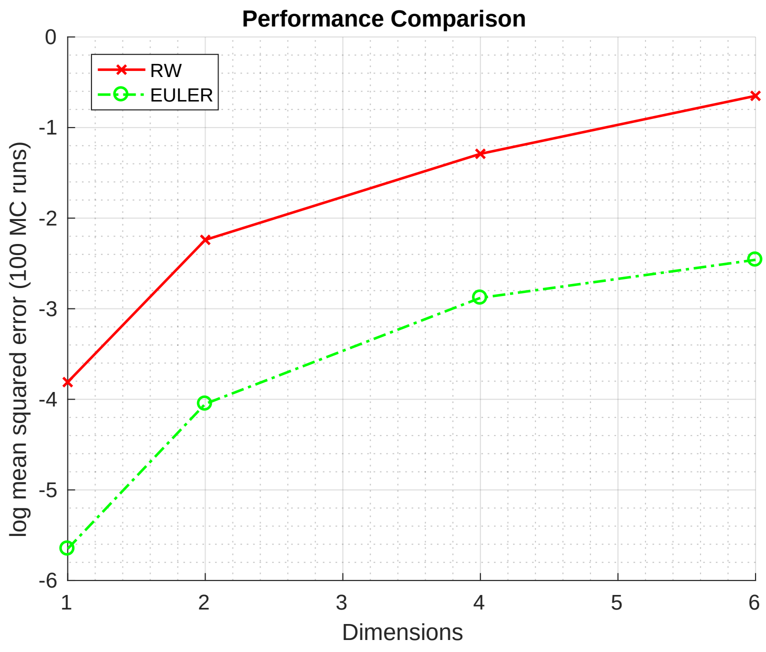


FIGURE 6.8: Comparison of the random walk and the Euler discretisation based on the log mean squared error (estimation of the mean value). Figure 6.7 illustrates the target distribution

6.4 Conclusions

The benefits of using the Euler discretisation in the proposal of the SMC sampler is discussed in this chapter. Using information from the derivative in the proposal distribution allows the algorithm to explore the space of interest more efficiently than the random walk proposal, which is used in the traditional SMC sampler. The efficiency improvement corresponds to better sampling process and as a result faster convergence to stationarity. The motivation for this strategy is inspired from research on identifying better proposals in the MH algorithm, such as the MALA. The results from this research reveal that considering a more sophisticated proposal distribution than the random walk leads to faster convergence in the context of SMC samplers while indicating a valuable direction for future research.

Chapter 7

Optimal backward kernel

7.1 Introduction

The backward Markov kernel, similarly with the forward Markov kernel, is a user-defined distribution. In Chapter 6, we discussed the benefits of using more sophisticated proposal distributions and how such choices can lead to faster convergence. Both methods influence the quality of the generated samples or particles during the importance sampling stage. In this chapter, an optimal backward Markov kernel is proposed with respect to the selected forward Markov kernel. This proposed method allows the overall algorithm to obtain better performance. In Section 7.2, a description of the novel optimal Markov backward kernel is discussed, and experiments are conducted in Section 7.4 to illustrate the benefits of the proposed method.

7.2 Optimal backward Markov kernel

In an SMC sampler, the target distribution is defined using a joint distribution parameterised by a backward Markov kernel, $L(x_{k-1}|x_k)$, such that the target is a density defined on the states $x_{1:k}$ as

$$\pi_{1:k}\left(x_{1:k}\right) = \pi_k\left(x_k\right)\prod_{i=2}^{k} L\left(x_{i-1}|x_i\right) \tag{7.1}$$

where $\pi_k\left(x_k\right)$ is the target distribution over the sequence of samples $x_{1:k}$ or joint distribution of all the states up to the kth state. This construction has the property that

$$\int \pi_{1:k}\left(x_{1:k}\right) dx_{1:k-1} = \pi_k\left(x_k\right) \tag{7.2}$$

such that if we draw samples that target $\pi_{1:k}\left(x_{1:k}\right)$, then the marginal distribution (for x_k) of these samples is $\pi_k\left(x_k\right)$ [66] [37]. At the kth iteration, the ith (of N samples) is then associated with a state, $x_k^{(i)}$, and a weight, $w_k^{(i)}$, where

$$w_k^{(i)} = w_{k-1}\frac{\pi_k\left(x_k\right)}{\pi_k\left(x_{k-1}\right)}\frac{L\left(x_{k-1}|x_k\right)}{q\left(x_k|x_{k-1}\right)} \tag{7.3}$$

corresponds to the importance weights and is practically expressed in logarithmic scale to avoid numerical issues. A choice for the backward Markov kernel is to emulate MCMC (i.e., the importance weights computation is similar to the MH ratio [72]) as $L(x_{k-1}|x_k) = q(x_k|x_{k-1})$, where $L(.)$ and $q(.)$ denote the backward and forward Markov kernels, respectively. Such a choice is poor or inefficient in most cases and results in importance weights with very large or infinite variance [72]. Appendix B.3 and [68] mention that the proposal distribution should be heavier tailed compared to the target distribution, and $\frac{\pi(x)}{q(x)} > c$, where c is a positive constant. If the target distribution is heavier tailed than the proposal, then the particle weights approach infinity (i.e., in Equation 7.3, $w_k^{(i)} \to \infty$). A function of interest $\bar{f}$ at the kth iteration is estimated as

$$\bar{f} \approx \hat{f}_k = \sum_{i=1}^{N} w_k^{(i)} f\left(x_k^{(i)}\right) \tag{7.4}$$

As further explained in Appendix B.1 the $\hat{f}_k$ is an unbiased estimator of $\bar{f}$.

To understand how to choose $L\left(x_{k-1}|x_k\right)$, we need to consider the variance of an estimator of a function, $f\left(x_k\right)$. Based on the argument in Appendix B.2, the variance will be dependent on

$$a_L\left(L\left(x_{k-1}|x_k\right)\right) = \int \frac{\pi\left(x_k\right)^2 L\left(x_{k-1}|x_k\right)^2 f\left(x_k\right)^2}{q\left(x_k|x_{k-1}\right) q\left(x_{k-1}\right)} dx_k dx_{k-1} \tag{7.5}$$

We wish to find the $L\left(x_{k-1}|x_k\right)$ that minimises $a\left(L\left(x_{k-1}|x_k\right)\right)$ subject to the constraint that $L\left(x_{k-1}|x_k\right)$ is a probability density function (pdf) for every state x_k. In other words, we wish to minimise

$$\begin{aligned} b_L\left(L\left(x_{k-1}|x_k\right)\right) = \int \frac{\pi\left(x_k\right)^2 L\left(x_{k-1}|x_k\right)^2 f\left(x_k\right)^2}{q\left(x_k|x_{k-1}\right) q\left(x_{k-1}\right)} dx_k dx_{k-1} - \\ \int \lambda_{x_k}^L \left(\int L\left(x_{k-1}|x_k\right) dx_{k-1}\right) dx_k \end{aligned} \tag{7.6}$$

where $\lambda_{x_k}^L$ is a Lagrangian multiplier for a specific value of x_k. To minimise the function we can differentiate $b_L\left(L\left(x_{k-1}|x_k\right)\right)$ and set to zero (unique solution)

$$\frac{db_L\left(L\left(x_{k-1}|x_k\right)\right)}{dL\left(x_{k-1}|x_k\right)} = 0 \tag{7.7}$$

such that

$$2\frac{\pi\left(x_k\right)^2 L_{opt}\left(x_{k-1}|x_k\right) f\left(x_k\right)^2}{q\left(x_k|x_{k-1}\right) q\left(x_{k-1}\right)} - \lambda^L_{x_k} = 0 \tag{7.8}$$

By noting that we can cater for everything that does not depend on x_{k-1} in the normalisation constant, Equation 7.8 implies that

$$L_{opt}\left(x_{k-1}|x_k\right) \propto q\left(x_k|x_{k-1}\right) q\left(x_{k-1}\right) \tag{7.9}$$

$$= \overleftarrow{q}\left(x_{k-1}|x_k\right) \tag{7.10}$$

where it is worth noting that, in general:

$$\overleftarrow{q}\left(x_{k-1}|x_k\right) \neq q\left(x_{k-1}|x_k\right) \tag{7.11}$$

Here $q\left(x_{k-1}|x_k\right)$ is the probability density associated with a proposal defined at x_k sampling x_{k-1} whereas $\overleftarrow{q}\left(x_{k-1}|x_k\right)$ is the probability density associated with the proposal having been defined at x_{k-1} given that it resulted in sampling x_k.

7.3 Near optimal backward Markov kernel

While $\overleftarrow{q}\left(x_{k-1}|x_k\right)$ is, in general, difficult to calculate analytically, we can approximate this optimal $L\left(x_{k-1}|x_k\right)$ by exploiting the fact that we have samples from $q\left(x_k, x_{k-1}\right)$. Specifically, we propose to use the samples in the filter to estimate the parameters of some parametric density (e.g., Gaussian) that approximates $q\left(x_k, x_{k-1}\right)$. Using this parametric approximation, we can then deduce an approximation to $\overleftarrow{q}\left(x_{k-1}|x_k\right)$.

7.3.1 Parametric estimation of the joint density

For example, if we assume that $q\left(x_k, x_{k-1}\right)$ is well approximated as Gaussian, we can use a Kalman-filter update to estimate $\overleftarrow{q}\left(x_{k-1}|x_k\right)$ as follows. If

$$q\left(x_k, x_{k-1}\right) \approx \mathcal{N}\left(\begin{bmatrix} x_k \\ x_{k-1} \end{bmatrix}; \begin{bmatrix} \mu_k \\ \mu_{k-1} \end{bmatrix}, \begin{bmatrix} \Sigma_{k,k} & \Sigma_{k,k-1} \\ \Sigma_{k-1,k} & \Sigma_{k-1,k-1} \end{bmatrix}\right) \tag{7.12}$$

then

$$\overleftarrow{q}\left(x_{k-1}|x_k\right) = \mathcal{N}\left(x_{k-1}; \mu_{k-1|k}, \Sigma_{k-1|k}\right) \tag{7.13}$$

where

$$\mu_{k-1|k} = \mu_{k-1} + \Sigma_{k-1,k} \left(\Sigma_{k,k}\right)^{-1} \left(x_k - \mu_k\right) \tag{7.14}$$

$$\Sigma_{k-1|k} = \Sigma_{k-1,k-1} - \Sigma_{k-1,k} \left(\Sigma_{k,k}\right)^{-1} \Sigma_{k,k-1} \tag{7.15}$$

7.3.2 Baseline method

One method for approximating $q\left(x_k, x_{k-1}\right)$ would be to use the samples, $x_{k-1:k}^{(i)}$, directly such that

$$\mu_{k_1} = \frac{1}{N} \sum_{i=1}^{N} x_{k_1}^{(i)} \tag{7.16}$$

$$\Sigma_{k_1,k_2} = \frac{1}{N-1} \sum_{i=1}^{N} \left(x_{k_1}^{(i)} - \mu_{k_1}\right) \left(x_{k_2}^{(i)} - \mu_{k_2}\right)^T \tag{7.17}$$

for $k_1 \in \{k-1, k\}$ and $k_2 \in \{k-1, k\}$.

7.3.3 Avoiding resampling errors

In [72], it is proposed that if resampling has occurred at the k^{th} iteration, then we can approximate as

$$q\left(x_{k-1}, x_k\right) \approx \pi\left(x_{k-1}\right) q\left(x_k | x_{k-1}\right) \tag{7.18}$$

Indeed, we could use the ancestral samples of x_{k-1} associated with each of the current samples of x_k. However, it is well known that since even when using a variant that minimises the variance, resampling introduces what can be thought of as quantisation errors in the weights (i.e. after resampling the weights are always $\frac{1}{N}$, while this is not true before the resampling. This is illustrated in Figure 7.1).

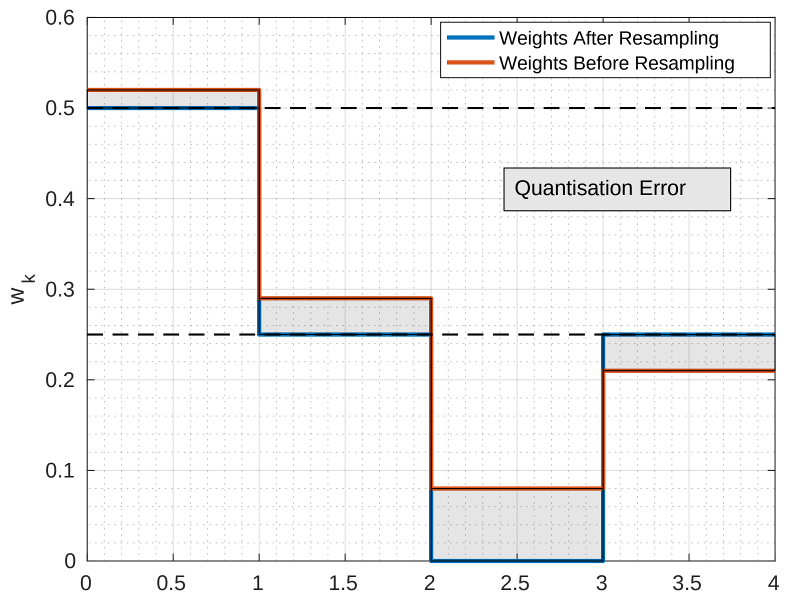


FIGURE 7.1: Exemplar of the quantisation errors introduced in the resampling algorithm

So, we adopt a different approach by asserting that it is preferable to use the samples of x_{k-1} that were available at the $(k-1)$th iteration to estimate the parametric approximation to $q\left(x_{k-1}, x_k\right)$. Therefore, we approximate as

$$q\left(x_{k-1}, x_k\right) \approx \sum_{i=1}^{N} w_{k-1}^{(i)} q\left(x_k | x_{k-1}^{(i)}\right) \delta\left(x_{k-1} - x_{k-1}^{(i)}\right) \tag{7.19}$$

We can then calculate the parameters of a Gaussian approximation to $q\left(x_{k-1}, x_k\right)$ as

$$\mu_{k-1} = \sum_{i=1}^{N} \tilde{w}_{k-1}^{(i)} x_{k-1}^{(i)} \tag{7.20}$$

$$\mu_k = \sum_{i=1}^{N} \tilde{w}_{k-1}^{(i)} \mu\left(x_{k-1}^{(i)}\right) \tag{7.21}$$

$$\Sigma_{k-1,k-1} = \frac{1}{N^\star} \sum_{i=1}^{N} \tilde{w}_{k-1}^{(i)} \left(x_{k-1}^{(i)} - \mu_{k-1}\right) \left(x_{k-1}^{(i)} - \mu_{k-1}\right)^T \tag{7.22}$$

$$\Sigma_{k,k-1} = \frac{1}{N^\star} \sum_{i=1}^{N} \tilde{w}_{k-1}^{(i)} \left(\mu\left(x_{k-1}^{(i)}\right) - \mu_k\right) \left(x_{k-1}^{(i)} - \mu_{k-1}\right)^T \tag{7.23}$$

$$\Sigma_{k,k-1} = {\Sigma_{k,k-1}}^T \tag{7.24}$$

$$\Sigma_{k,k} = \Sigma_q^2 + \frac{1}{N^\star} \sum_{i=1}^{N} \tilde{w}_{k-1}^{(i)} \left(\mu\left(x_{k-1}^{(i)}\right) - \mu_k\right) \left(\mu\left(x_{k-1}^{(i)}\right) - \mu_k\right)^T \tag{7.25}$$

where $\mu\left(x_{k-1}^{(i)}\right)$ is the mean of $q(x_k^{(i)}|x_{k-1}^{(i)})$ and $q(x_k|x_{k-1}) = \mathcal{N}(x_k, \mu(x_{k-1}), \Sigma_q^2)$, $\tilde{w}_{k-1}^{(i)}$ are the weights input to the resampling process and where (from [84])

$$\frac{1}{N^{\star}} = \frac{\sum_{i=1}^{N} \tilde{w}_{k-1}^{(i)}}{\left(\sum_{i=1}^{N} \tilde{w}_{k-1}^{(i)}\right)^2 - \sum_{i=1}^{N} \left(\tilde{w}_{k-1}^{(i)}\right)^2}. \tag{7.26}$$

which becomes the familiar Bessel's correction (i.e., pre-multiplication by $\frac{1}{N-1}$, not $\frac{1}{N}$) in the case where $\tilde{w}_{k-1}^{j} = \frac{1}{N}$ for all i.

7.4 Simulation results

The simulations focus on sampling from one-dimensional static distributions and estimating the true mean value. The first set of simulations in 7.4.1 compare the optimal and traditional backward kernels. The second set of simulations in 7.4.2 compare the SMC sampler using the optimal backward kernel and competitor methodologies.

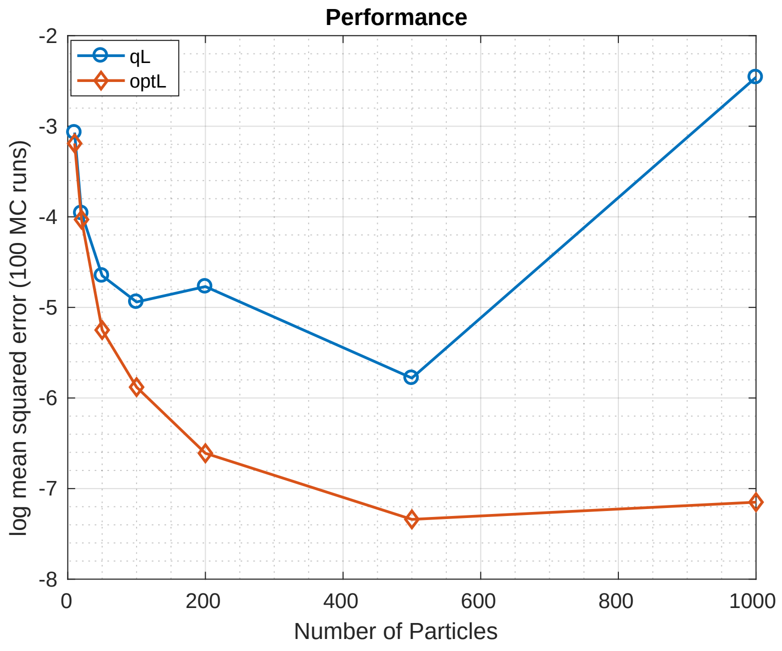
7.4.1 Comparison of SMC sampler with optimal and basic backward Markov kernels

Three static distributions are used for the experiments the Gaussian distribution with mean value 2 and covariance 1, the Student's t-distribution with degrees of freedom 7 and the Laplace distribution. The estimation of the true mean value is based on the average value over 100 Monte Carlo runs. The total number of particles varies, but in all cases there are in total 10000 samples (the multiplication of the number of iterations in the SMC sampler and the number of particles is always equal to 10000). A list of different scenarios are examined:

1. Comparison of the SMC sampler with optimal backward kernel and the basic SMC sampler, where both methods use random walk proposal and without recycling. This is the simplest scenario where the only improvement in the SMC sampler is the backward Markov kernel. The results in Table 7.1 and Figures 7.2 7.3 7.4 indicate that for small number particles both methods have similar behaviour, while the best performance is performed from the optimal backward Markov kernel for 500 particles.

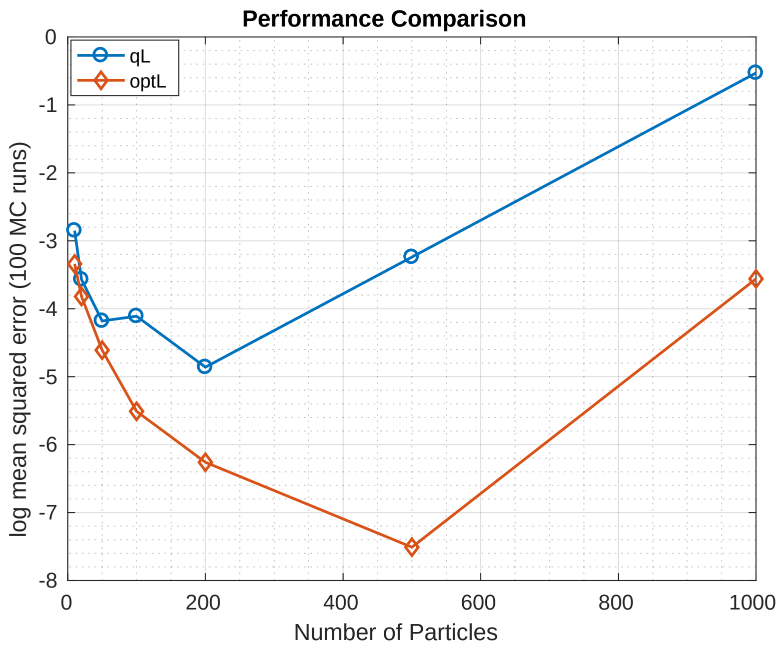
TABLE 7.1: Comparison of the two SMC samplers based on the last iteration's log mean squared error (estimation of the mean value).

	Gaussian		**Student's-t**		**Laplace**	
Par. × It.	qL	optL	qL	optL	qL	optL
10 × 1000	-3.07	-3.19	-2.85	-3.34	-2.54	-2.95
20 × 500	-3.96	-4.03	-3.57	-3.82	-3.27	-4.02
50 × 200	-4.65	-5.25	-4.18	-4.61	-3.39	-5.06
100 × 100	-4.94	-5.88	-4.11	-5.51	-4.00	-5.45
200 × 50	-4.77	-6.61	-4.86	-6.26	-4.15	-5.93
500 × 20	-5.78	-7.34	-3.24	-7.51	-2.46	-6.73
1000 × 10	-2.46	-7.15	-0.53	-3.56	-0.78	-2.36



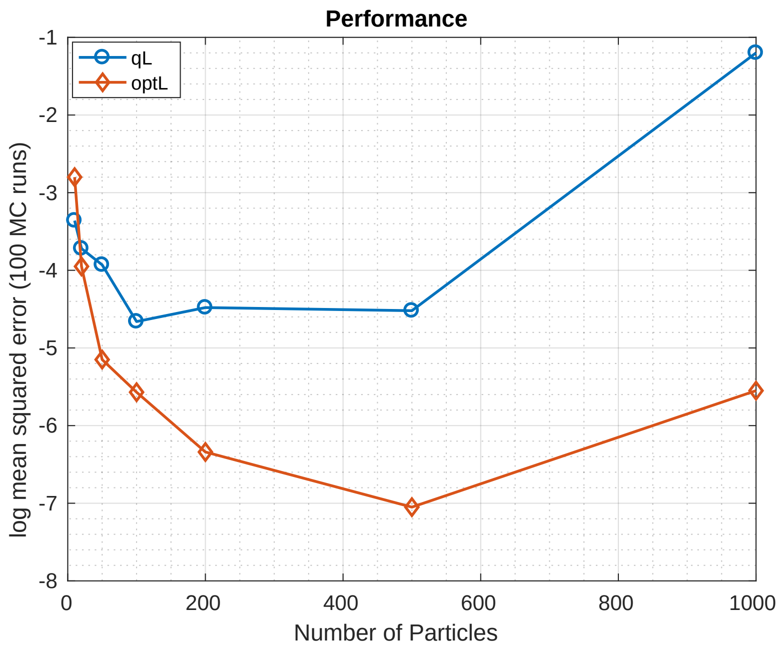
(a) Gaussian

FIGURE 7.2: Graphical presentation of the Table 7.1. Every point corresponds to the same total number of samples from the Gaussian target distribution.



(b) Student’s-t

FIGURE 7.3: Graphical presentation of the Table 7.1. Every point corresponds to the same total number of samples from the Student’s-t target distribution.



(c) Laplace

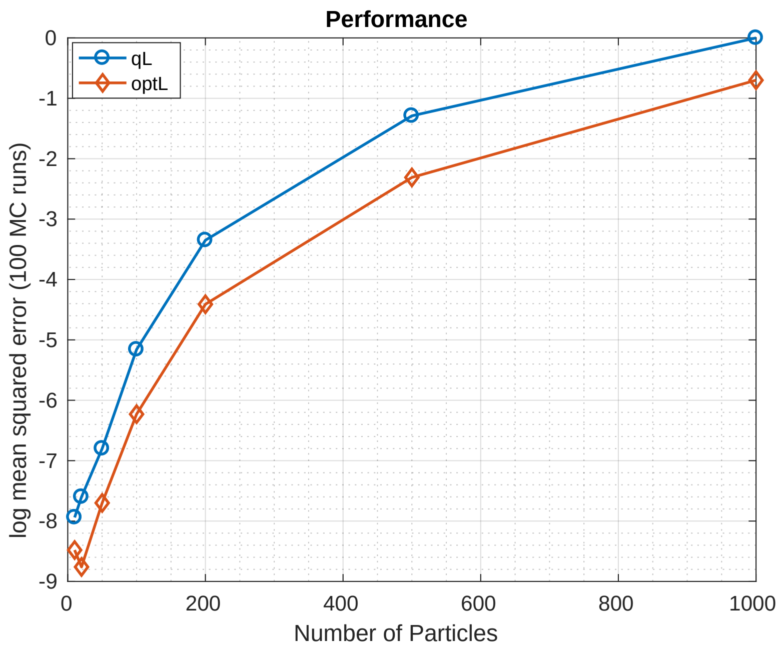
FIGURE 7.4: Graphical presentation of the Table 7.1. Every point corresponds to the same total number of samples from the Laplace target distribution.

2. Comparison of the SMC sampler with optimal backward kernel and the basic

SMC sampler, where both methods use random walk proposal and the basic recycling method 5. This scenario is identical with the previous except for the recycling, which is activated for both backward Markov kernels. In Table 7.2 and Figures 7.5 7.6 7.7, the SMC sampler with optimal backward Markov kernel outperforms in all cases with its best performance when applying a relatively small number of particles with a larger number of iterations. It is noticeable that when the number of particles is very small, then the accuracy might become worse as there are not enough particles to describe the distribution. The ideal number of particles and iterations will depend on the application or posterior distribution.

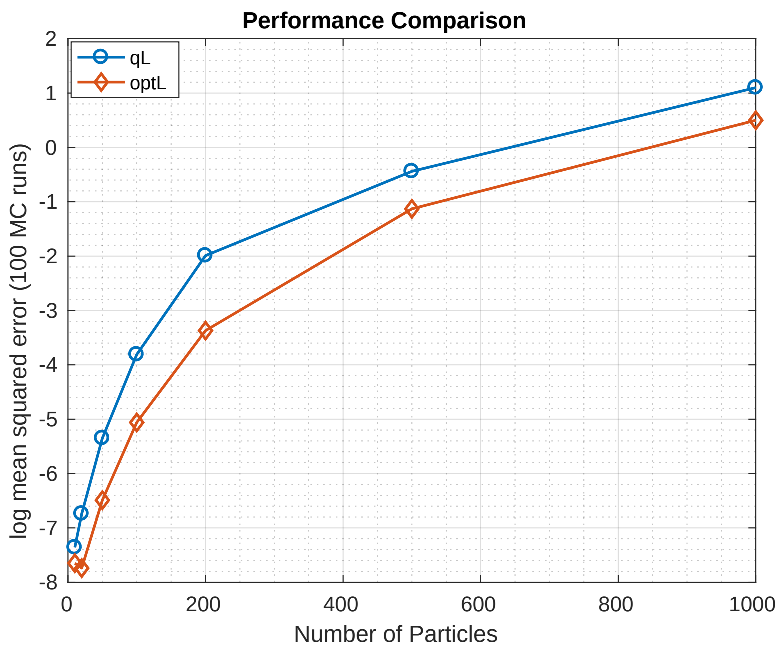
TABLE 7.2: Comparison of the two SMC samplers based on the last iteration's log mean squared error (estimation of the mean value).

	Gaussian		**Student's-t**		**Laplace**	
Par. × It.	qL	optL	qL	optL	qL	optL
10 × 1000	-7.94	-8.48	-7.36	-7.65	-6.22	-7.17
20 × 500	-7.60	-8.76	-6.74	-7.74	-5.93	-7.37
50 × 200	-6.80	-7.7	-5.35	-6.49	-4.75	-6.42
100 × 100	-5.16	-6.23	-3.8	-5.06	-3.38	-4.96
200 × 50	-3.35	-4.41	-1.99	-3.37	-1.78	-2.96
500 × 20	-1.29	-2.31	-0.44	-1.14	-0.17	-0.96
1000 × 10	0.00	-0.7	1.10	0.50	0.59	0.38



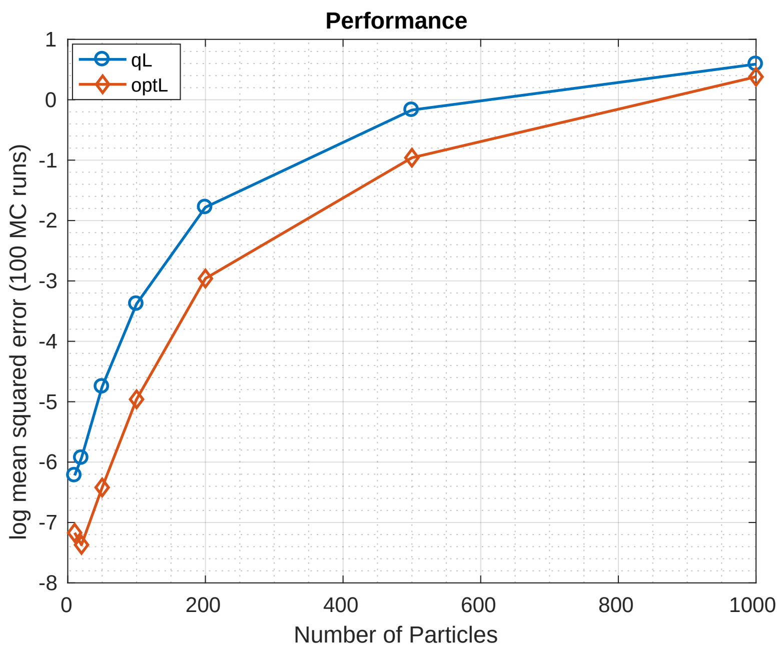
(a) Gaussian

FIGURE 7.5: Graphical presentation of the Table 7.2. Every point corresponds to the same total number of samples from the Gaussian target distribution.



(b) Student's-t

Figure 7.6: Graphical presentation of the Table 7.2. Every point corresponds to the same total number of samples from the Student's-t target distribution.



(c) Laplace

Figure 7.7: Graphical presentation of the Table 7.2. Every point corresponds to the same total number of samples from the Laplace target distribution.

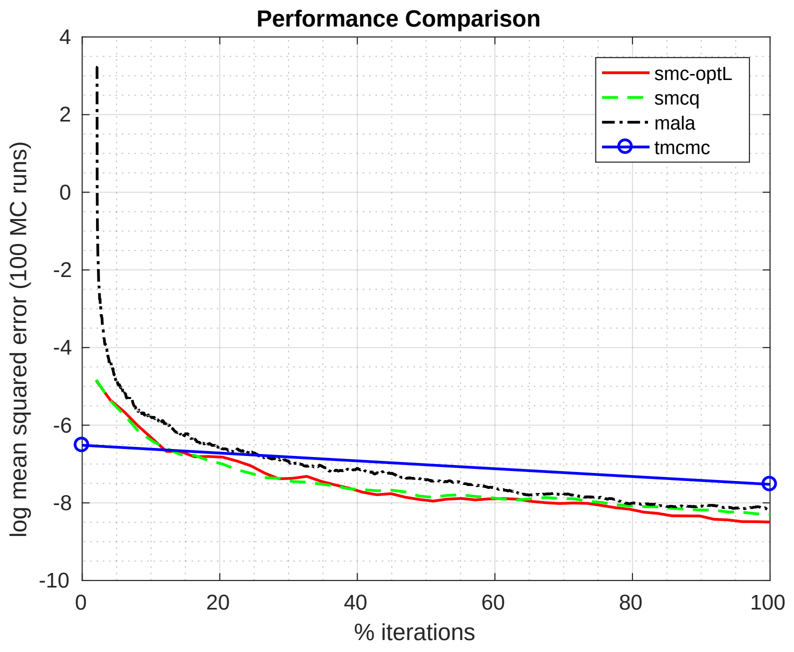
7.4.2 Comparison of the SMC sampler with optimal backward Markov kernel with competitor methodologies

The second part of the evaluation is the comparison of the proposed SMC sampler with competitor approaches, including the Metropolis adjusted Langevin algorithm (MALA), the Transitional Markov Chain Monte Carlo (TMCMC), and the SMC sampler using the traditional backward kernel, Euler proposal, and the recycling method. Two scenarios of a unimodal (Gaussian) distribution and multimodal (Gaussian mixture) distribution are examined.

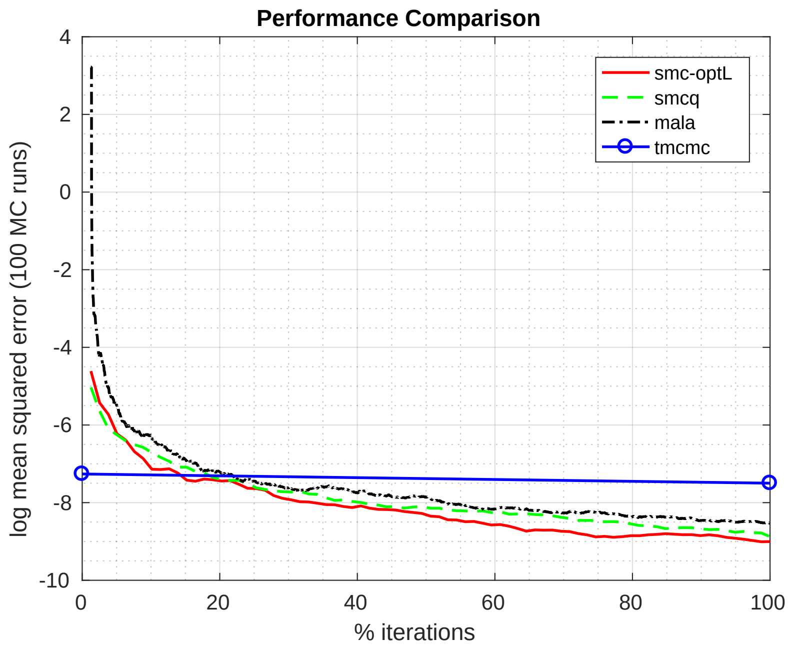
7.4.2.1 Comparison on a unimodal distribution

The first argument is based on the time complexity and parallelisation of the competing methods. The SMC sampler algorithm is a fully distributed algorithm while the TMCMC requires sequential computations. For example, in TMCMC, the annealing schedule and chains update during every iteration is executed sequentially. The parallel complexity of the TMCMC algorithm is $O(N)$.

The performance of all methods depends on the total number of samples. The number of iterations in the TMCMC algorithm is fixed (i.e., not a user-defined parameter) and depends on the simulated annealing (i.e., intermediate distributions until convergence). The number of iterations on the MALA and SMC samplers is a user-defined parameter. Theoretically, the MALA and SMC sampler using a fixed number of samples or particles can be executed continuously. For example, consider a Gaussian distribution with a true mean value of 5 and covariance of 1. We apply both algorithms to generate samples for estimating the true mean value. In the first case, Figure 7.8, we consider 1000 samples for the TMCMC, while we execute the other algorithms for longer times. In the SMC samplers, we consider 100 particles in all cases.



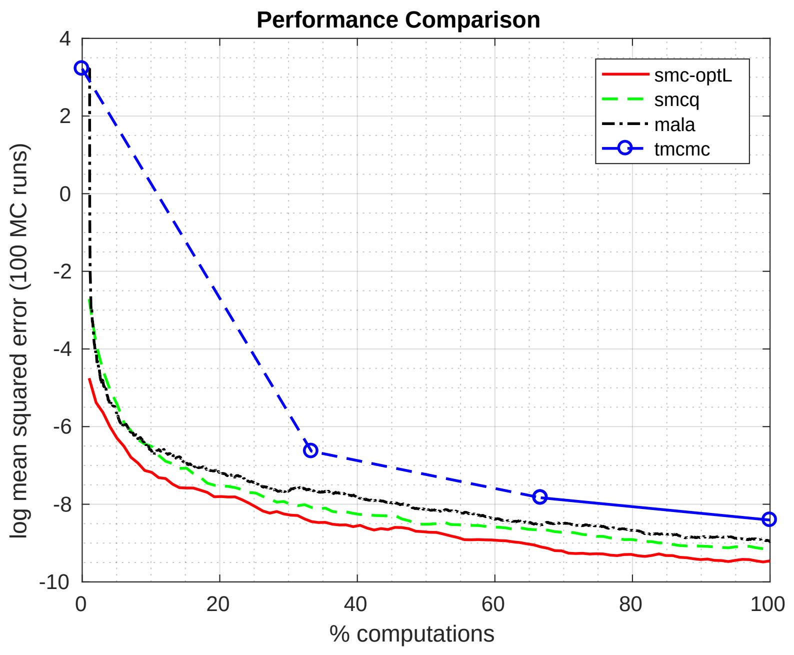
(a) TMCMC: 1000 samples, MALA/SMC samplers: 5000 samples



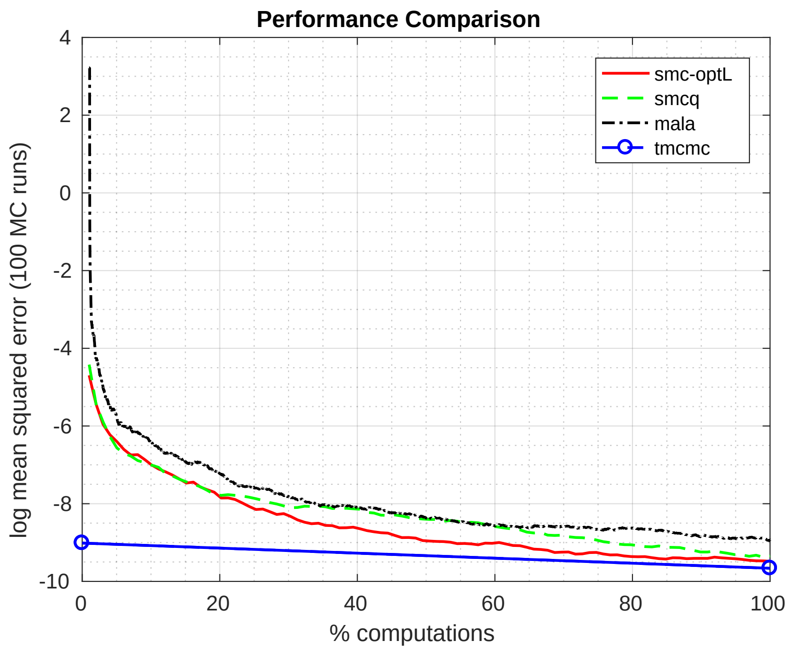
(b) TMCMC: 1000 samples, MALA/SMC samplers: 8000 samples

FIGURE 7.8: Performance comparison of the four methods. We consider 1,000 samples for the TMCMC algorithm, while for the other methods the algorithms can continue running independently of the number of initial samples.

The performance of all methods is dependent on the initial proposal. A wider proposal will affect the convergence of all methods as they require more iterations to reach equilibrium. In such a scenario, the TMCMC algorithm requires more intermediate distributions until convergence. For instance, consider a Gaussian distribution with a mean value of 5 and covariance of 1. For the comparison two different initialisation for the algorithms: (Case A) Uniformly distributed numbers in the interval (-20, 20), and (Case B) Uniformly distributed numbers in the interval (2, 8). In the former case, presented in Figure 7.9, the TMCMC converges after the creation of three intermediate distributions, while in the later case only one.



(Case A) Samples initialised uniformly in $(-20, 20)$



(Case B) Samples initialised uniformly in $(2, 8)$

FIGURE 7.9: Performance comparison of the four methods generating 10000 samples from the posterior. The step size is one in the MALA and SMC sampler algorithms.

7.4.2.2 Comparison on a bimodal distribution

This simulation runs sampling for estimating the true mean value on a Gaussian mixture. The two scenarios considered are an equally weighted Gaussian mixture of $0.5N(20,1)+0.5N(50,1)$ with a true mean value of 35, and a non-equally weighted Gaussian mixture of $0.8N(20,1)+0.2N(50,1)$ with true mean value 26. All algorithms generate 10000 samples. The SMC sampler is applied using 100 particles and 100 iterations.

If the distance between the mean values of the two modes is considerable, then the SMC sampler will fail to converge. More precisely, an unforeseen issue is caused by the resampling algorithm. The particles from one mode become more dominant than those of the other mode. After a few iterations, all particles are in one of the two modes. As a result, the algorithm samples and performs estimations only on this one mode, while the other mode is ignored. The resampling encountering this issue cannot be avoided as it provides the solution to the degeneracy phenomenon. Two approaches are proposed as a solution to the problem, which focus on the proposal during the importance sampling step of the algorithm.

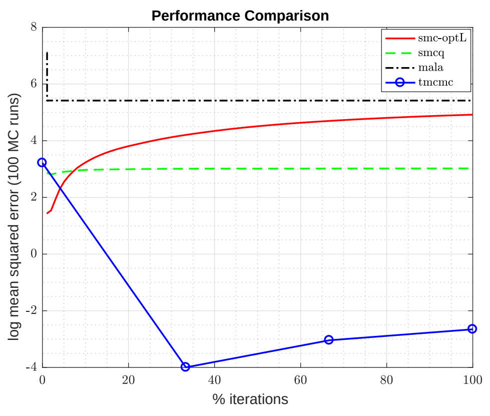
1. The first approach is named **"Fixed" particles** (Algorithm 14) where for every iteration, the state is "fixed" according to the initial proposal distribution. During the importance sampling step, the particles are located in the two modes.

Algorithm 14 "Fixed" Particles

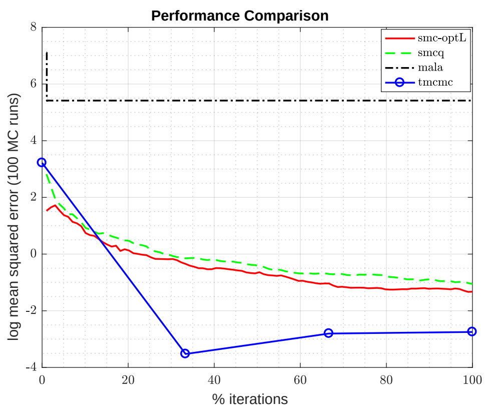
```
1: for k do=2:K
2:     ▷ User defined integers $\epsilon$, $\alpha$ and $\beta$, where $\beta > \alpha$
3:     Set $x_{k-1}$ uniformly in $(\alpha, \beta)$
4:     $x_k \sim q(x_k|x_{k-1}^{(i)}) = \mathcal{N}(x_{k-1}^{(i)}, \sqrt{\epsilon})$, with $i = 1, \ldots, N$ number of particles
5:     $q(x_k|x_{k-1}) = \beta \in \mathcal{R}^{Nx1}$ (i.e. flat for all particles)
6:     ...
7: end for
```

2. The second approach is the named **Particles Grouping** where during its second iteration of the algorithm, it is the same as the "fixed" particles algorithm. At the end of the second iteration the particles are grouped using k-means. In the following iterations, particles are generated using the centroids as the mean value of the heavier-tailed Gaussian distributions. Half of the particles are used to create a Gaussian with the mean value the first centroid and the other half for the second.

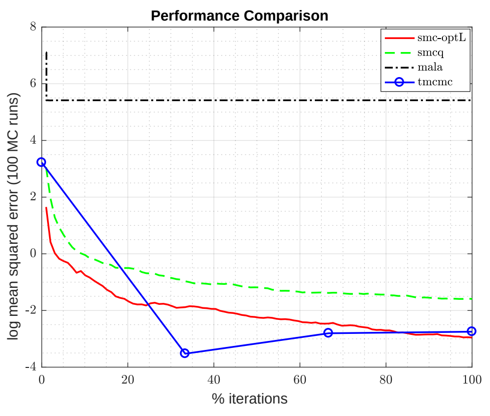
For both solutions, the problem is not solved but hidden. A direct solution to the problem would require reconsideration of the computation of the resampling algorithm. In Figures 7.10 and 7.12, the performance of the SMC sampler using the different methodologies is provided and compared with the TMCMC and MALA algorithms. Exemplars of the samples generated in the TMCMC, MALA and SMC sampler with optimal L-kernel are available in Figures 7.11 and 7.13 for the corresponding Figure 7.10 and 7.12, respectively.



(a) Initial comparison

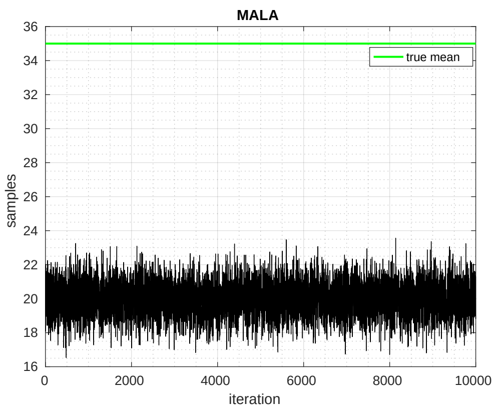


(b) "Fixed" particles method

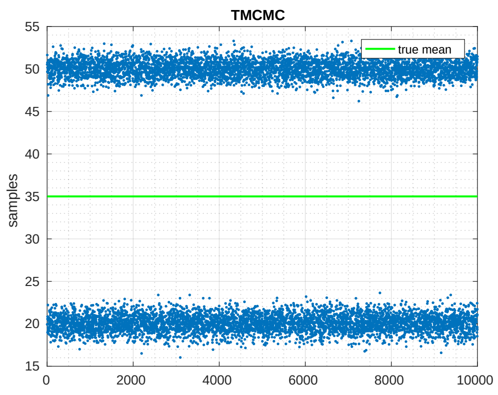


(c) Particles Grouping method

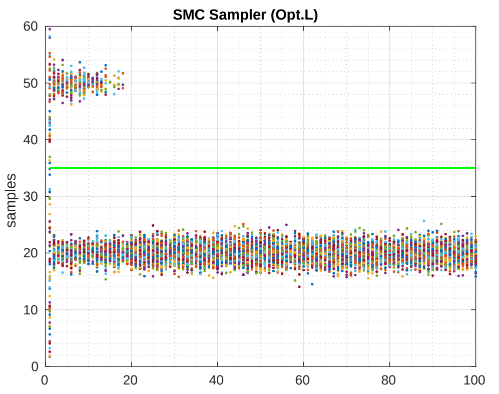
FIGURE 7.10: (a) comparison based on the SMC samplers with optimal backward Markov kernel, (b) comparison using the "Fixed" particles method and (c) comparison using the particles grouping method.



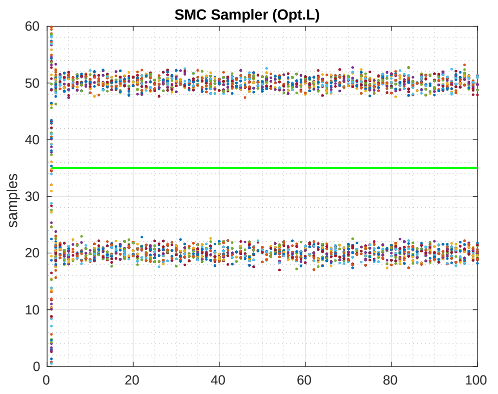
(a) MALA samples.



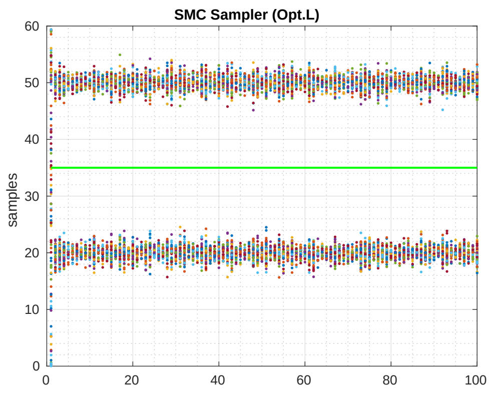
(b) TMCMC samples.



(c) Basic algorithm.

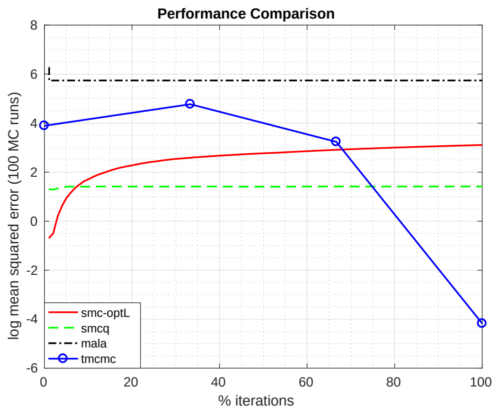


(d) "Fixed" paricles method.

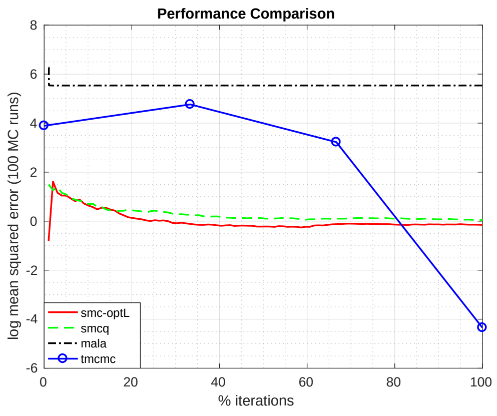


(e) Particles grouping method.

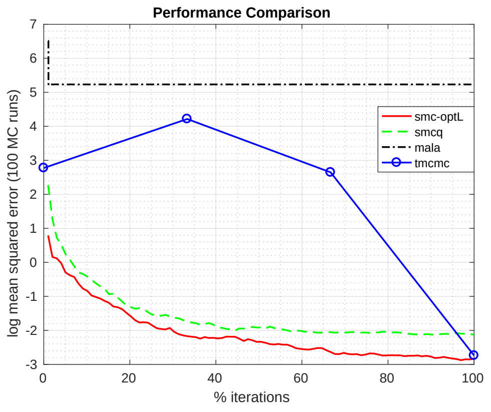
FIGURE 7.11: Exemplar of the samples generated in a single Monte Carlo run for the algorithms (a) TMCMC, (b) MALA, (c-e) SMC sampler with optimal L-kernel in the three different methodologies. See Figure 7.10 for the performance comparison.



(a) Basic SMC sampler.

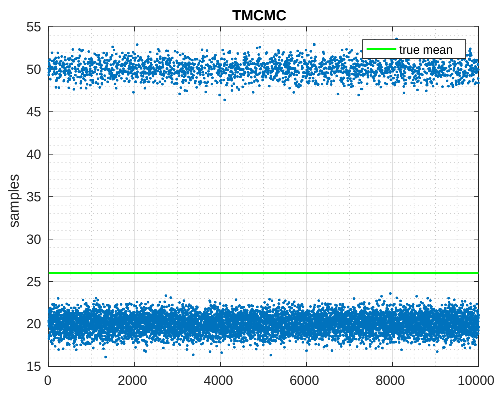


(b) "Fixed" particles method.

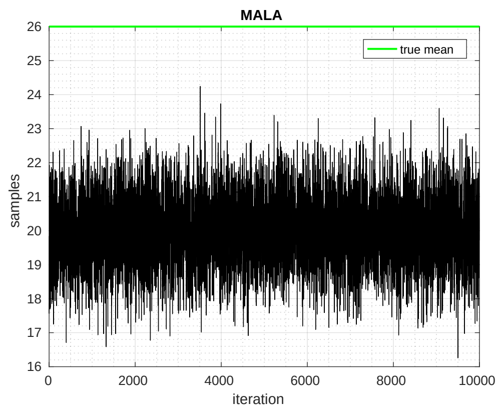


(c) Particles grouping method

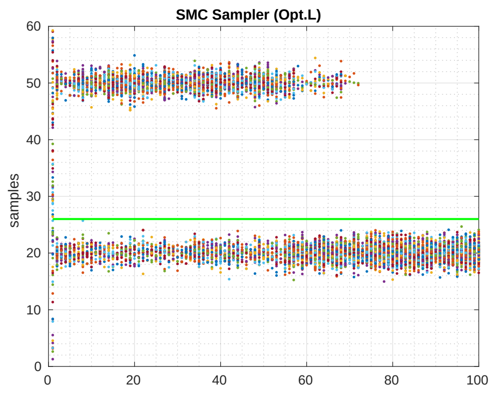
FIGURE 7.12: (a-c) Performance comparison of the 4 methods using different methodologies in the SMC samplers.



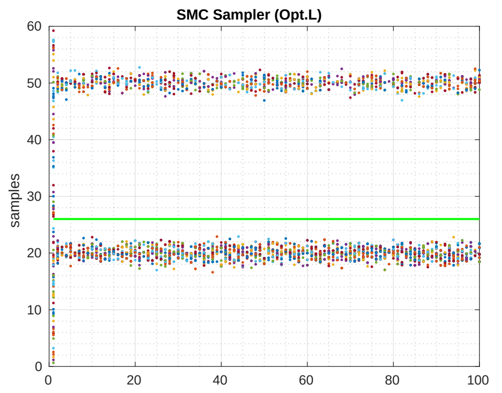
(a) TMCMC samples.



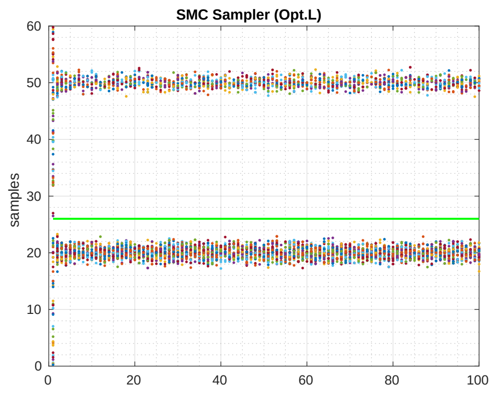
(b) MALA samples.



(c) Basic SMC sampler.



(d) "Fixed" particles method.



(e) Particles grouping method.

FIGURE 7.13: Exemplar of the samples generated in a single Monte Carlo run for the algorithms (a) TMCMC, (b) MALA, (c-e) SMC sampler with optimal L-kernel in the three different methodologies. See Figure 7.12 for the performance comparison.

7.5 Conclusions

A novel optimal backward kernel was proposed for the SMC sampler, and the performance of the new method is compared with the traditional algorithm and competitor methods. These simulations demonstrate the potential of the new method and the benefits it offers to the performance of the SMC sampler. Future research will include a performance comparison on high dimensional spaces and provide methodologies to overcome the issue in multimodal distributions.

Chapter 8

Conclusions

A new method is proposed to train the Radial Basis Function (RBF) network where the RBF centres are updated using steps of the importance sampling and resampling. In the original method, the core algorithm to update the RBF centres is the Metropolis-Hastings (MH). A comparison of the two methods reveals that the proposed method does not perform as well as the original algorithm. This result relates to the initial comparison of the traditional Sequential Monte Carlo (SMC) and Markov chain Monte Carlo (MCMC) methods, so it is difficult for the original SMC sampler to outperform the MH algorithm.

Based on the above observations, the research emphasises the accuracy and performance improvements of the SMC sampler. The proposed method is fully distributed with better accuracy over the original SMC sampler. A fully distributed SMC method is required for two reasons. First, increasing the number of particles leads to a better representation of the probability density function and accuracy improvement. As a result, it needs to use as many particles as possible. Second, increasing the number of particles leads to increasing the computational time. A detailed analysis of the proposed distributed method is provided, and the new method was applied in Big Data frameworks and High Performance Computing (HPC). Future research will focus on the application of the method in hardware-based implementations.

The accuracy improvement of the original SMC sampler is based on several strategies. First, a new method to combine estimates over multiple iterations (or particles recycling) was discussed. Second, a strategy to consider more sophisticated proposal distributions was suggested. In the original algorithm, the proposal density is a random walk, while better accuracy can be achieved with Langevin-based proposal distributions. It was observed from the initial comparison of the traditional SMC and MCMC methods that the Hamiltonian Monte Carlo (HMC) algorithm outperforms all the methodologies significantly. A future direction will concentrate on the benefits of using Hamiltonian dynamics as the proposal in the SMC sampler. Finally, a near optimal backward Markov kernel was applied and compared to the original SMC sampler.

Incorporating these strategies together creates the proposed and improved SMC sampler, which was compared with the original algorithm and competitor MCMC methodologies.

Appendix A

Parallelising particle filters with deterministic runtime on distributed memory systems

A.1 Introduction

In this section, we reformulate the new proposed redistribute algorithm with improved time complexity (Chapter 4), and the rest of the particle filter components, for the distributed memory setup. As such, we repeat the experiments from Chapter 4 to demonstrate this environment is more suitable than MapReduce for the algorithm. In Section A.2, we describe a distributed memory system followed by the implementation aspects of these components on distributed memory architectures in Section A.3. Section A.5 represents the results and analysis of the implementations with directions for further research in Section A.6.

A.2 Distributed memory systems

A distributed memory system is constructed from multiple independent computational nodes interconnected by a high-speed network. Each node is equipped with memory that is only addressable by the cores or processors within that node. This arrangement is different to a shared memory system where both computational and memory spaces are confined to a single node. Accesses to memory spaces, and, thus, to the data by computational units (or cores) can be local or remote. Remote access to data is facilitated by explicitly sending and receiving data, referred to as messages, between the computational units owning the data. The Message Passing Interface (MPI) provides the means for handling communication between cores on distributed memory systems by uniquely identifying the cores through the assignment of unique identifiers, known as ranks. Such explicit communication and computational models lead to the notion of data ownership and scalability. However, the disadvantages include the cost of communication and the associated data movements, which may affect the overall speedups, especially on inefficient implementations.

A.3 MPI particle filter

In this section, we discuss the MPI implementation of the main particle filter components. These algorithms are designed for P MPI cores working in parallel. All data structures, such as the particles, x, the weights, w, and the array of the number of copies, *nCopies*, have N elements and are equally distributed over the MPI cores. In other words, each MPI core is allowed to access $n = \frac{N}{P}$ elements.

A.3.1 MPI cumulative sum

Modern MPI libraries provide a built-in function, called MPI Scan, to perform the cumulative sum. However, if the number of MPI cores $P < N$, then each core must perform a local sequential sum before calling the MPI Scan and the series of subsequent subtractions.

Pseudocode of the parallel MPI Cumulative Sum is described by Algorithm 15. We can infer that the time complexity is equal to $O\left(\frac{N}{P} + \log_2 P\right)$ which converges to $O(\log_2 N)$ when $P = N$.

Algorithm 15 MPI Cumulative Sum

```
Input: $N$, $P$, $x$
Output: $y$
1: $n \leftarrow \frac{N}{P}$
2: $local\_sum \leftarrow 0$
3: for $i \leftarrow 0$; $i < n$; $i \leftarrow i + 1$ do
4:     $local\_sum \leftarrow local\_sum + x^i$
5: end for
6: MPI_Scan($local\_sum, ...$)
7: for $i \leftarrow n - 1$; $i \geq 0$; $i \leftarrow i - 1$ do
8:     $temp \leftarrow x^i$
9:     $y^i \leftarrow local\_sum$
10:    $local\_sum \leftarrow local\_sum - temp$
11: end for
```

A.3.2 MPI Bitonic sort

Bitonic sort is one of the fastest parallel sorting algorithms. The achieved time complexity is $O\left(N(\log_2 N)^2\right)$ for a single core implementation and $O\left((\log_2 N)^2\right)$ with $P = N$ processors working in parallel [16]. For our purposes, we need a modified version of Bitonic sort because, while we sort *nCopies*, the particles will consequently move.

The pseudocode for a possible MPI implementation is described by Algorithm 16. As can be seen, each MPI process selects a new partner during each iteration. Next, the partners send each other both nCopies and particles as well as locally call the Bitonic Merge. This means that the number of sent messages is equal to $O\left((\log_2 P)^2\right)$.

Algorithm 16 MPI Bitonic Sort

Input: $nCopies$, x, N, P, $rank$

Output: $nCopies$, x

```
1: $n \leftarrow \frac{N}{P}$
2: Serial Bitonic Sort$(nCopies, x, n)$
3: for $i \leftarrow 2$; $i \leq P$; $i \leftarrow 2 \cdot i$ do
4:     $up \leftarrow$ Direction$(rank, i)$
5:     for $j \leftarrow 0$; $j < \log_2 i$; $j \leftarrow j + 1$ do
6:         $partner \leftarrow$ Partner Calc$(rank, i, j)$
7:         MPI_Sendrecv$(nCopies, ...)$
8:         MPI_Sendrecv$(x, ...)$
9:         Bitonic_Merge$(nCopies, x, ..., up, n)$
10:     end for
11: end for
```

A.3.3 MPI minimum variance resampling

MVR is the first step of the Multinomial resampling and aims to minimise the ergodic variance of the new population of particles [67]. To do this, we first calculate the cumulative sum of the weights in a similar way as described in Section A.3.1. Each $nCopies^i$ is then calculated independently with a for loop whose iteration space gets smaller when the number of cores P increases. The overall time complexity is then equal to $O(\log_2 N)$ when $P = N$.

Algorithm 17 MPI MVR

Input: N, P, $rank$, w

Output: $nCopies$

```
1: $n \leftarrow \frac{N}{P}$
2: if $rank == 0$ then
3:     $c^0 \leftarrow 0$
4: end if
5: $[c^1, ..., c^N] \leftarrow$ MPI Cumulative Sum$(N, P, w)$
6: for $i \leftarrow 0$; $i < n$; $i \leftarrow i + 1$ do
7:     $nCopies^i \leftarrow floor\left(c^{i+1}\right) - floor\left(c^i\right)$
8: end for
```

A.4 MPI redistribute

In this section, we discuss how to implement on MPI the three Redistribute algorithms previously considered throughout this thesis.

A.4.1 MPI $O(N)$ redistribute

Algorithm 18 describes a possible implementation of the Redistribute algorithm, which is called naive implementation in [97].

Since $nCopies$ satisfies the following property:

$$\sum_{i=0}^{N-1} nCopies^i = N \tag{A.1}$$

Algorithm 18 $O(N)$ Redistribute

Input: N, $nCopies$, x

Output: x_{new}

```
1: for $j \leftarrow 0$; $j < N$; $j \leftarrow j + 1$ do
2:     for $k \leftarrow 0$; $k < nCopies_j$; $k \leftarrow k + 1$ do
3:         $x^i_{new} \leftarrow x^i$
4:         $i \leftarrow i + 1$
5:     end for
6: end for
```

it can be inferred that Algorithm 18 achieves $O(N)$ time complexity. Although this algorithm has a very low time constant and is very fast on a single core, it is notoriously difficult to obtain an efficient parallel implementation. This is because the workload solely depends on the contents of $nCopies$, which is runtime dependent. As such, the workload can become extremely unbalanced depending on the contents of $nCopies$. On distributed memory architectures, parallelisation is further complicated by the partitioned memory spaces are. Although MPI-specific techniques, such as all-to-all communication routines, can be used to provide easier data access across partitions, the time complexity would still be $O(N)$ even with $P = N$ cores. Furthermore, the runtime is likely to be worse than a single core implementation with the added communication costs.

The proposed MPI implementation for this algorithm addressing these issues is described in Algorithm 19 and then re-distributes the particles to the other cores. All-to-one and one-to-all communication routines are necessary to gather and distribute the particles.

Algorithm 19 MPI $O(N)$ Redistribute

Input: N, $rank$, $nCopies$, x

Output: x

```
1: MPI_Gather(...)
2: if $rank == 0$ then
3:     $temp\_x \leftarrow O(N)$ Redistribute($N, nCopies, x$)
4: end if
5: MPI_Scatter($x, ..., temp\_x$)
```

A.4.2 MPI $O\left((\log_2 N)^3\right)$ redistribute

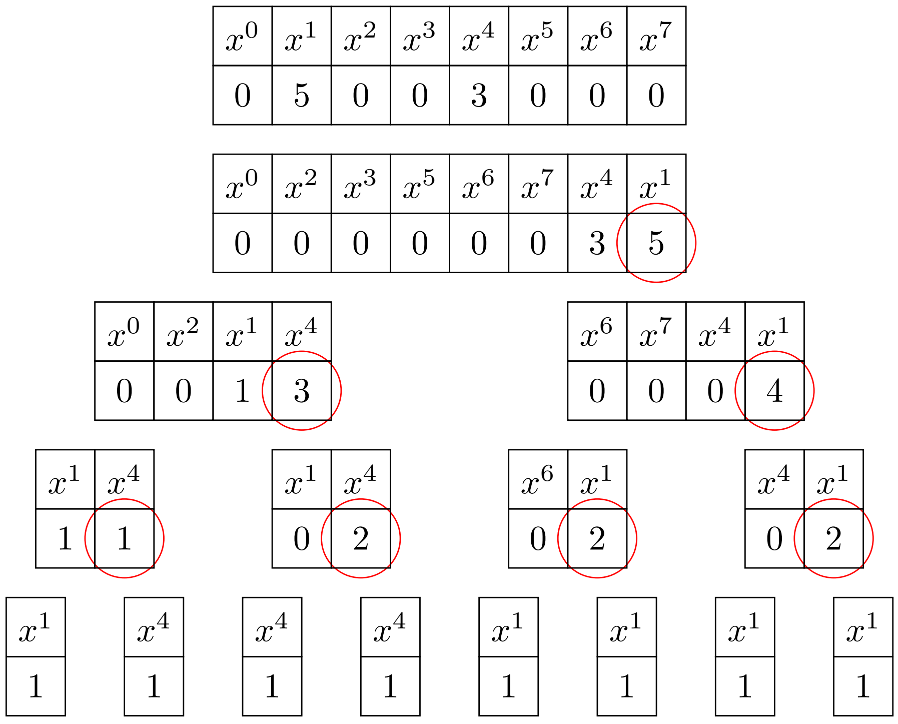


Figure A.1: $O\left((\log_2 N)^3\right)$ Redistribute

Figure A.1 shows an example of the $O\left((\log_2 N)^3\right)$ Redistribute, whose pseudocode is described by Algorithm 20. Bitonic sort is the first task of every stage of the binary tree. This step is necessary to divide the workload deterministically since the particles would be randomly distributed otherwise. The cumulative sum is then performed to calculate the position of the pivot, which is circled in red in Figure A.1.

The function Distribute splits and distributes the particles on either side of the pivot. Each core is coupled with another core of the same node. One core acts as a sender, and its partner acts as a receiver, so that the particles move from the right side to the left of the node. If *nCopies* is not sorted, each sender-receiver pair cannot be calculated deterministically, and this step would take more than $O(1)$ operations.

In the last step, we call MPI Comm split to generate as many new communicators as the number of nodes we have in the following stage of the binary tree. This recursive routine stops when the size of the root node $N = n$. At this point, each MPI core calls the $O(N)$ Redistribute locally.

Algorithm 20 MPI $O\left((\log_2 N)^3\right)$ Redistribute

Input: $Node = [nCopies, x]$, N, P, n, $rank$

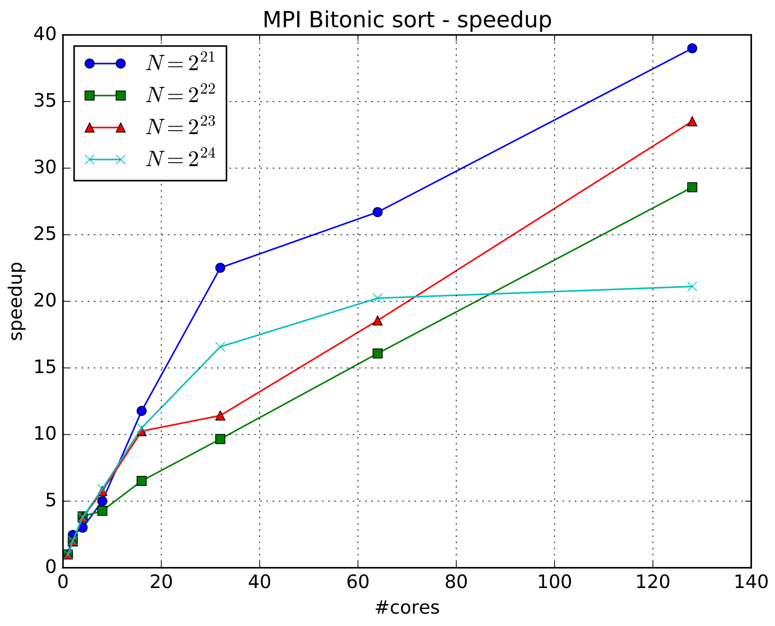
Output: x

```
1: if $N == n$ then
2:     $x \leftarrow O(N)$ Redistribute$(n, rank, nCopies, x)$
3:     return $x$
4: end if
5: MPI Bitonic Sort$(Node, N, P, rank)$
6: $csum \leftarrow$ MPI Cumulative Sum$(N, P, nCopies)$
7: $pivot \leftarrow$ Pivot Calc$(nCopies, csum)$
8: $(Leaf_l, Leaf_r) \leftarrow$ Distribute$(Node, csum, pivot)$
9: $P \leftarrow \frac{P}{2}$
10: $N \leftarrow \frac{N}{2}$
11: $colour \leftarrow (int)(\frac{rank}{P})$
12: MPI_Comm_split$(..., colour, rank, ...)$
13: MPI_Comm_size$(...)$
14: MPI_Comm_rank$(...)$
15: $O\left((\log_2 N)^3\right)$ Redistribute$(Leaf_l, N, P, n, rank)$
16: $O\left((\log_2 N)^3\right)$ Redistribute$(Leaf_r, N, P, n, rank)$
```

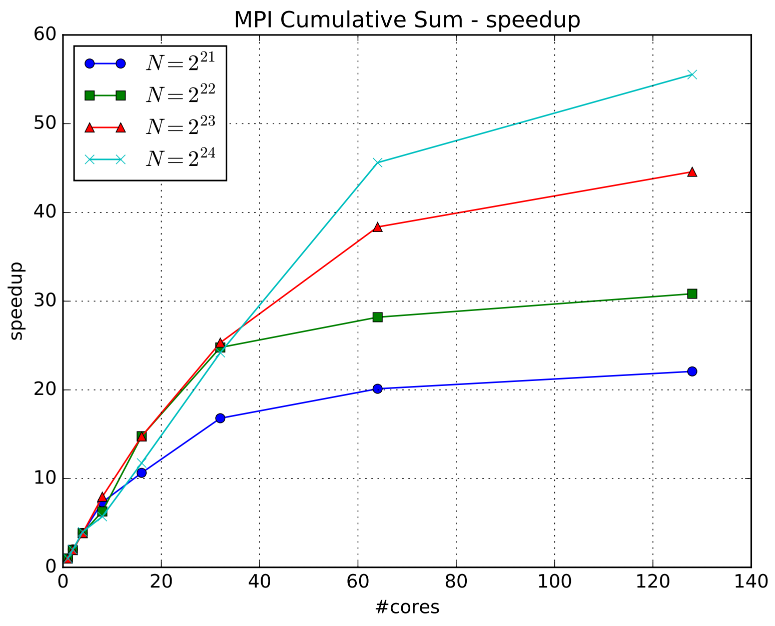
A.4.3 MPI $O\left((\log_2 N)^2\right)$ redistribute

Figure A.3 shows an example of the $O\left((\log_2 N)^2\right)$ Redistribution applied to the same of example of Figure A.1 and Algorithm 21 describes the pseudocode.

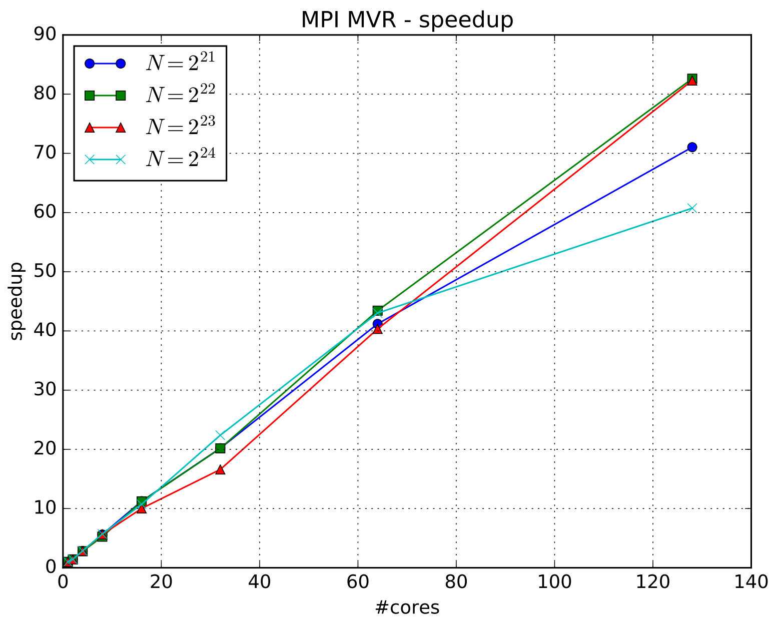
As stated in previous sections, we only need to sort the particles once before the algorithm descends the binary tree. In the binary tree phase, Bitonic sort is replaced by rotational shifts to ensure the workload is still distributed deterministically.



(a) Speedup: MPI-Cumulative Sum



(b) Speedup: MPI Bitonic Sort



(c) Speedup: MPI MVR

Figure A.2: Figures for basic algorithmic components

Table A.1: Tables for basic algorithmic components

(a) Runtimes:
MPI-Cumulative Sum (ms)

P	N			
	2^{21}	2^{22}	2^{23}	2^{24}
1	4.99	10.33	20.97	42.15
2	2.47	5.235	10.78	21.17
4	1.288	2.68	5.426	10.55
8	0.68	1.64	2.636	7.367
16	0.469	0.699	1.421	3.585
32	0.297	0.417	0.828	1.74
64	0.248	0.366	0.547	0.924
128	0.226	0.335	0.47	0.759

(b) Runtimes: Bitonic Sort (s)

P	N			
	2^{21}	2^{22}	2^{23}	2^{24}
1	0.854	1.6	3.486	7.573
2	0.432	0.808	1.743	3.728
4	0.284	0.415	0.925	1.994
8	0.171	0.374	0.605	1.28
16	0.073	0.245	0.34	0.721
32	0.038	0.166	0.305	0.457
64	0.032	0.099	0.188	0.374
128	0.022	0.056	0.104	0.357

(c) Runtimes: MVR (s)

P	N			
	2^{21}	2^{22}	2^{23}	2^{24}
1	0.2	0.401	0.802	1.616
2	0.141	0.284	0.572	1.135
4	0.071	0.144	0.283	0.565
8	0.036	0.076	0.142	0.283
16	0.018	0.036	0.08	0.151
32	0.01	0.02	0.048	0.072
64	0.005	0.009	0.02	0.038
128	0.003	0.005	0.01	0.0266

Algorithm 21 MPI $O\left((\log_2 N)^2\right)$ Redistribute

Input: $Node = [nCopies, x]$, N, P, $n, rank$

Output: x

```
1: MPI Bitonic Sort($Node, N, P, rank$)
1: procedure BinaryTree($Node, N, P, n, rank$)
2:     if $N == n$ then
3:         $x \leftarrow O(N)$ Redistribute($n, rank,$
   $nCopies, x$)
4:         return $x$
5:     end if
6:     $csum \leftarrow$MPI Cumulative Sum($N, P, nCopies$)
7:     $local\_pivot \leftarrow$ Pivot Calc($nCopies, csum$)
8:     $pivot \leftarrow$ MPI_Allreduce($P, local\_pivot, 1, ...$)
9:     $r \leftarrow pivot - \left(\frac{N}{2} - 1\right)$
10:    $(Leaf_l, Leaf_r) \leftarrow$ Rot Shifts($Node, r$)
11:    $P \leftarrow \frac{P}{2}$
12:    $N \leftarrow \frac{N}{2}$
13:    $colour \leftarrow (int)(\frac{rank}{P})$
14:    MPI_Comm_split($..., colour, rank, ...$)
15:    MPI_Comm_size($...$)
16:    MPI_Comm_rank($...$)
17:    BinaryTree($Leaf_l, N, P, n, rank$)
18:    BinaryTree($Leaf_r, N, P, n, rank$)
19: end procedure
```

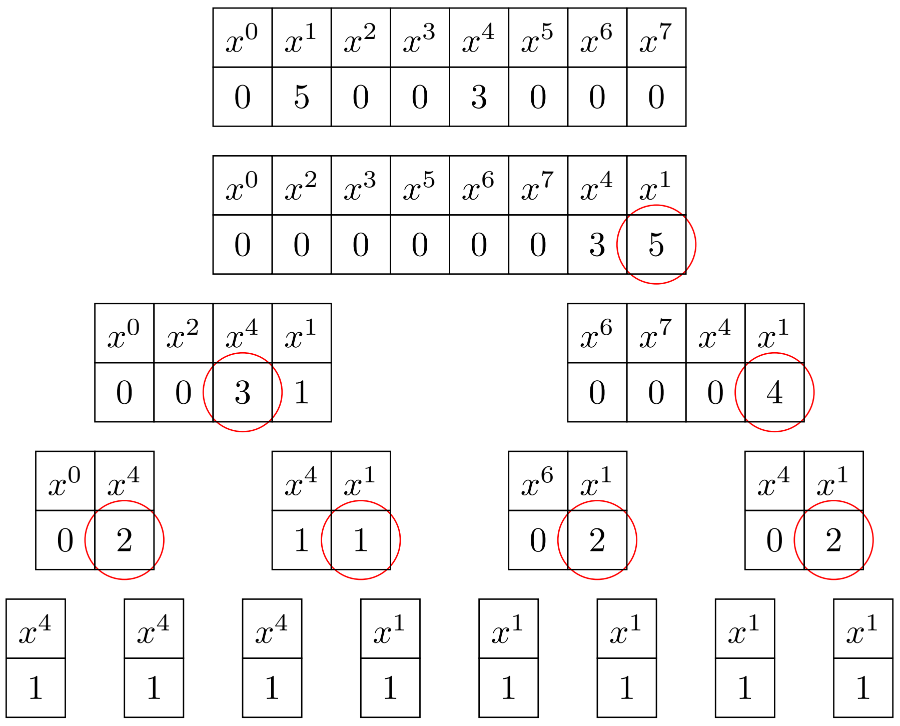


FIGURE A.3: $O\left((\log_2 N)^2\right)$ Redistribute

In the binary tree phase, cumulative sum is still performed stage-by-stage to calculate the position of the pivot. In this algorithm, it is necessary that all cores know the exact position of the pivot to calculate the number of rotations, r, that must be performed. Since the pivot could be located anywhere, we cannot use standard MPI communication routines, such as MPI Bcast, to broadcast the position of the pivot. We use MPI Allreduce instead. All cores but one will set the pivot to 0 and the MPI core owning the actual pivot will add the correct value. Once again, the communicators are split stage-by-stage until $N = n$, and each core will perform to linear time for the Redistribute described by Algorithm 18. Therefore, the binary tree step achieves $O\left((\log_2 N)^2\right)$ time complexity when $P = N$.

A.5 Evaluation

These MPI algorithms are tested for $N = 2^{21}, 2^{22}, 2^{23}, 2^{24}$ particles and for up to $P = 128$ MPI cores (see Table A.4 for the details of the system). All data structures, such as x, w and $nCopies$, are equally distributed all over the MPI cores, which means that each core owns $n = \frac{N}{P}$ particles. We provide results for runtime and speedup for P cores.

A.5.1 Cumulative sum, bitonic sort and MVR

Table A.1 and Figures A.2 show runtimes and speedups for Cumulative sum, Bitonic sort and MVR respectively. Cumulative Sum has been tested on random arrays of integers. MVR has been tested on random arrays of normalised floating point numbers which represented w. In order to test Bitonic sort, we used a random generator of arrays of integers which follow Equation A.1 and a random generator of arrays of floating point numbers to represent $nCopies$ and x respectively. However, the efficiency for $P = 128$ for Bitonic sort considerably decreases when N goes up, which is probably due to the higher percentage of cache misses.

A.5.2 Redistribute

Table A.2 and Figures A.7, A.8, A.9 show the results for all the three Redistribute algorithms we described in Section A.4. These algorithms are tested for the same random input. As can be seen, the $O(N)$. Redistribute is very fast for a few cores, but progressively becomes slower when P increases due to the increasing cost of communication. The runtimes for both the $O\left((\log_2 N)^2\right)$ and $O\left((\log_2 N)^3\right)$ Redistribute increase rapidly for $P = 2$ MPI cores, because neither Bitonic sort nor the binary tree phase are needed when $P = 1$. When the number of MPI cores goes up, we can see that the $O\left((\log_2 N)^2\right)$ Redistribute improves more quickly than the $O\left((\log_2 N)^3\right)$ and it eventually outperforms the linear time distribution for $P = 64$ or $P = 128$ MPI cores, depending on the dataset size.

Table A.2 indicates that many of the speedups are less than 1, which are not discussed here. Instead, more relevant results about the overall speedup for the particle filter are discussed in the following section.

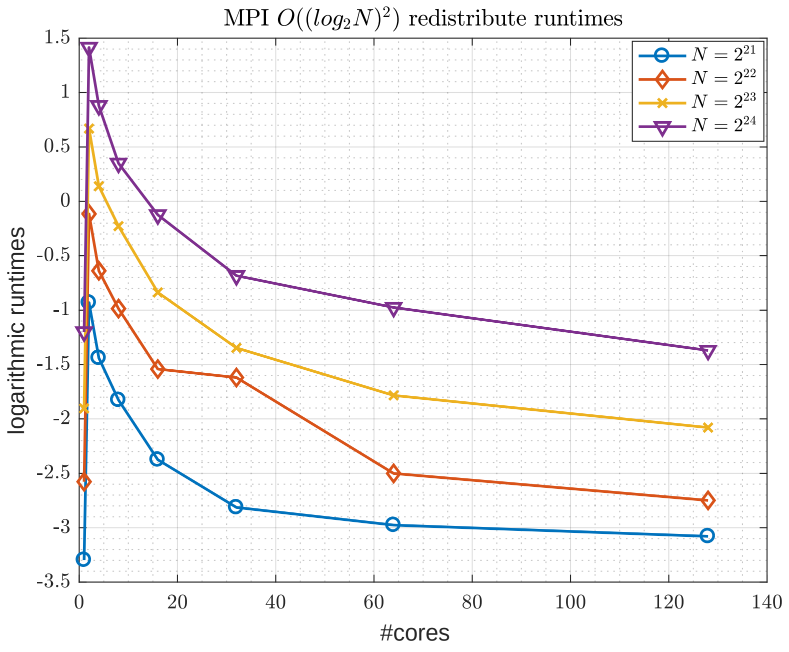


FIGURE A.4: $O\left((\log_2 N)^2\right)$ Redistribute runtimes

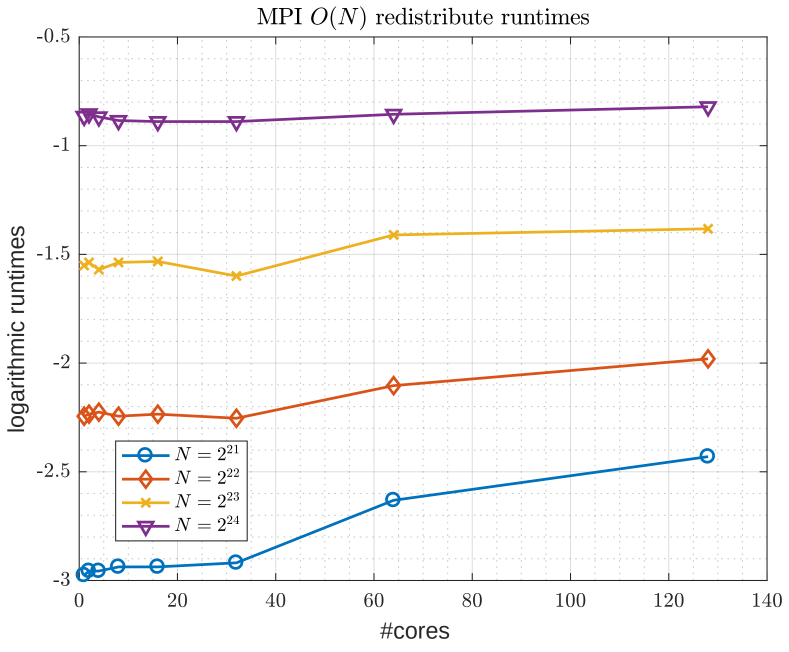


FIGURE A.5: $O\left(N\right)$ Redistribute runtimes

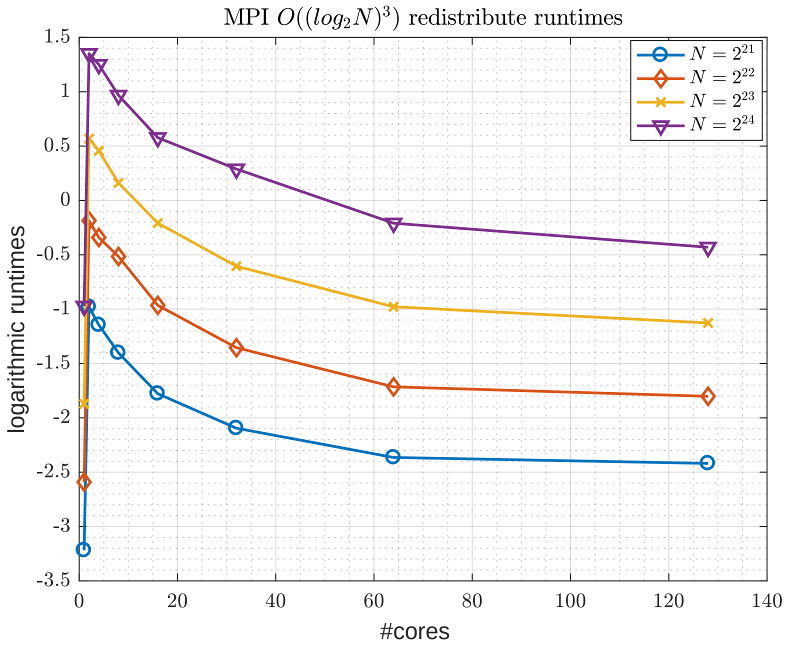


FIGURE A.6: $O\left((\log_2 N)^3\right)$ Redistribute runtimes

Table A.2: Runtimes: Redistribute (s)

			N			
			2^{21}	2^{22}	2^{23}	2^{24}
P	1	$O\left((\log_2 N)^2\right)$	0.037	0.076	0.149	0.301
		$O(N)$	0.051	0.106	0.212	0.421
		$O\left((\log_2 N)^3\right)$	0.04	0.075	0.154	0.378
	2	$O\left((\log_2 N)^2\right)$	0.394	0.891	1.952	4.114
		$O(N)$	0.052	0.107	0.215	0.426
		$O\left((\log_2 N)^3\right)$	0.376	0.83	1.765	3.86
	4	$O\left((\log_2 N)^2\right)$	0.237	0.528	1.149	2.409
		$O(N)$	0.052	0.108	0.208	0.42
		$O\left((\log_2 N)^3\right)$	0.318	0.711	1.579	3.49
	8	$O\left((\log_2 N)^2\right)$	0.161	0.373	0.796	1.419
		$O(N)$	0.053	0.106	0.215	0.413
		$O\left((\log_2 N)^3\right)$	0.246	0.597	1.175	2.63
	16	$O\left((\log_2 N)^2\right)$	0.093	0.214	0.433	0.883
		$O(N)$	0.053	0.107	0.216	0.411
		$O\left((\log_2 N)^3\right)$	0.169	0.381	0.812	1.781
	32	$O\left((\log_2 N)^2\right)$	0.06	0.198	0.26	0.505
		$O(N)$	0.054	0.105	0.202	0.411
		$O\left((\log_2 N)^3\right)$	0.123	0.258	0.546	1.335
	64	$O\left((\log_2 N)^2\right)$	0.051	0.082	0.168	0.377
		$O(N)$	0.072	0.122	0.244	0.425
		$O\left((\log_2 N)^3\right)$	0.094	0.18	0.376	0.811
	128	$O\left((\log_2 N)^2\right)$	0.046	0.064	0.125	0.254
		$O(N)$	0.088	0.138	0.251	0.44
		$O\left((\log_2 N)^3\right)$	0.089	0.165	0.324	0.65

A.5.3 Particle filter

In this section, we show the results for three versions of the SIR particle filters on MPI. Each runtime is taken for ten consecutive time steps, and to compare the algorithms accurately, we forced the worst case occurring when Redistribute is needed at every time step. A random Gaussian generator creates the array representing the state x.

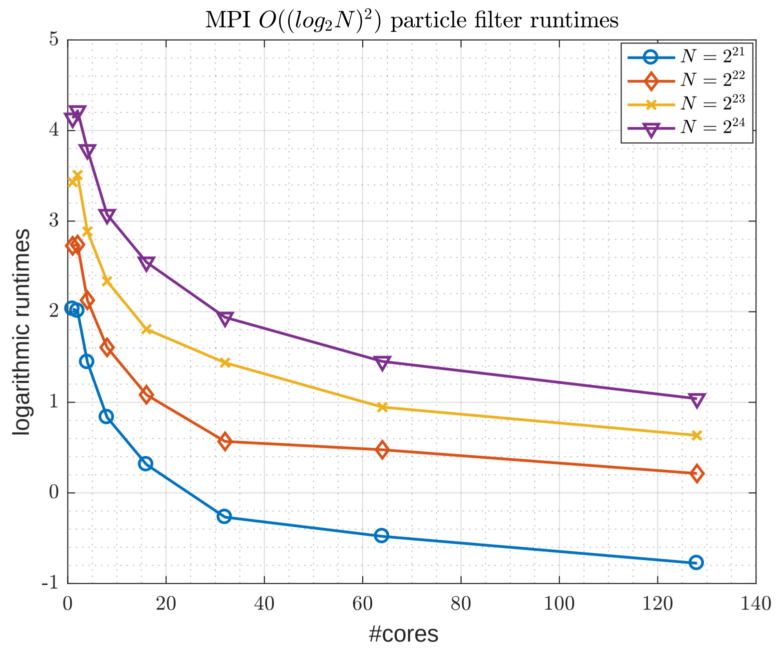


FIGURE A.7: $O\left((\log_2 N)^2\right)$ Particle filter runtimes

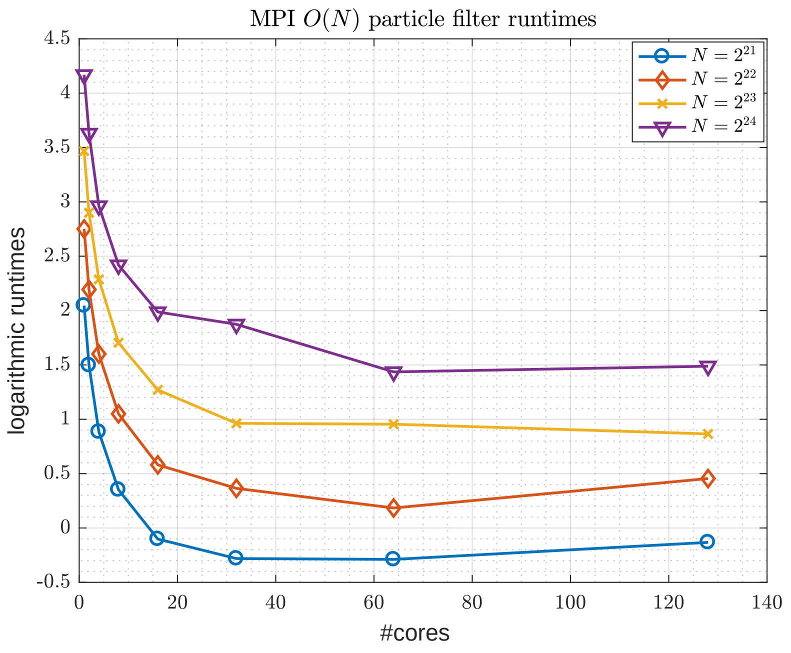


FIGURE A.8: $O(N)$ Particle filter runtimes

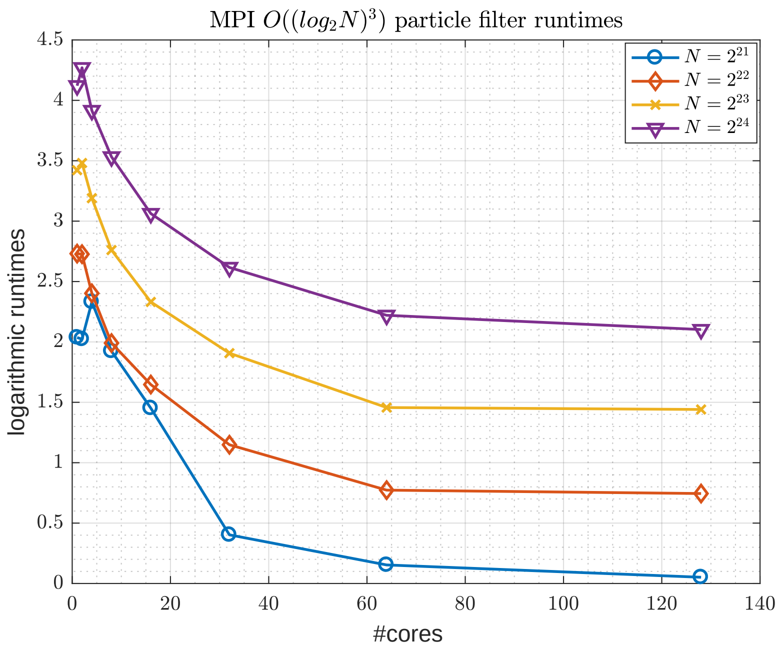


FIGURE A.9: $O\left((\log_2 N)^3\right)$ Particle filter runtimes

TABLE A.3: Runtimes: Overall Particle Filter (s)

			N			
			2^{21}	2^{22}	2^{23}	2^{24}
P	1	$O\left((\log_2 N)^2\right)$	7.617	15.3	30.86	62.57
		$O(N)$	7.733	15.66	32.05	64.64
		$O\left((\log_2 N)^3\right)$	7.664	15.35	30.63	61.63
	2	$O\left((\log_2 N)^2\right)$	7.456	15.51	33.44	67.9
		$O(N)$	4.468	8.967	18.17	37.7
		$O\left((\log_2 N)^3\right)$	7.56	15.28	32.5	71.54
	4	$O\left((\log_2 N)^2\right)$	4.234	8.389	17.93	44.2
		$O(N)$	2.422	4.957	9.848	19.35
		$O\left((\log_2 N)^3\right)$	10.29	11.05	24.29	50.2
	8	$O\left((\log_2 N)^2\right)$	2.304	4.985	10.35	21.62
		$O(N)$	1.422	2.858	5.495	11.23
		$O\left((\log_2 N)^3\right)$	6.846	7.319	15.81	34.21
	16	$O\left((\log_2 N)^2\right)$	1.369	2.954	6.093	12.78
		$O(N)$	0.903	1.787	3.562	7.295
		$O\left((\log_2 N)^3\right)$	4.27	5.192	10.29	21.39
	32	$O\left((\log_2 N)^2\right)$	0.765	1.765	4.209	6.946
		$O(N)$	0.755	1.44	2.618	6.513
		$O\left((\log_2 N)^3\right)$	1.496	3.154	6.733	13.69
	64	$O\left((\log_2 N)^2\right)$	0.619	1.61	2.577	4.269
		$O(N)$	0.749	1.202	2.598	4.204
		$O\left((\log_2 N)^3\right)$	1.166	2.166	4.292	9.212
	128	$O\left((\log_2 N)^2\right)$	0.46	1.239	1.887	2.829
		$O(N)$	0.876	1.576	2.376	4.429
		$O\left((\log_2 N)^3\right)$	1.053	2.106	4.224	8.186

Figures A.4 A.5, A.6 show the runtimes for each methodology on the overall particle filter. Table A.3 shows the speed of $O(N)$ particle filter improves for a limited number of cores. This behaviour is because the $O(N)$ Redistribute is faster than other tasks, such as MVR, when P is low. However, when P is high enough all tasks become faster than the $O(N)$ Redistribute. At this point, the $O(N)$ Redistribute emerges as the bottleneck and then the $O(N)$ Particle Filter stops scaling. On the other hand, the $O\left((\log_2 N)^2\right)$ and the $O\left((\log_2 N)^3\right)$ particle filter scale progressively for $P > 2$ cores. However, the most interesting result is that the $O\left((\log_2 N)^2\right)$ Particle Filter is not only faster than the $O\left((\log_2 N)^3\right)$ Particle Filter but, most importantly, it also outperforms the $O(N)$ particle filter for $P = 64, 128$, depending on N. These results prove that MPI is a better environment than MapReduce for the $O\left((\log_2 N)^2\right)$ Particle Filter. More precisely, the $O\left((\log_2 N)^2\right)$ particle filter is almost twice as fast as the $O(N)$ particle filter on MPI for $P = 128$ cores, while on MapReduce it was much slower for $P = 512$ cores.

Figure A.10 shows the speedups of the three particle filters for $N = 2^{24}$. The results for $N < 2^{24}$ are deducible from Table A.3 and are omitted for brevity. Figure A.10 underlines that the $O\left((\log_2 N)^2\right)$ particle filter for $P = 128$ is up to 22 times faster than each implementation of the particle filter for 1 core, as the three particle filter algorithms are equivalent when $P = 1$.

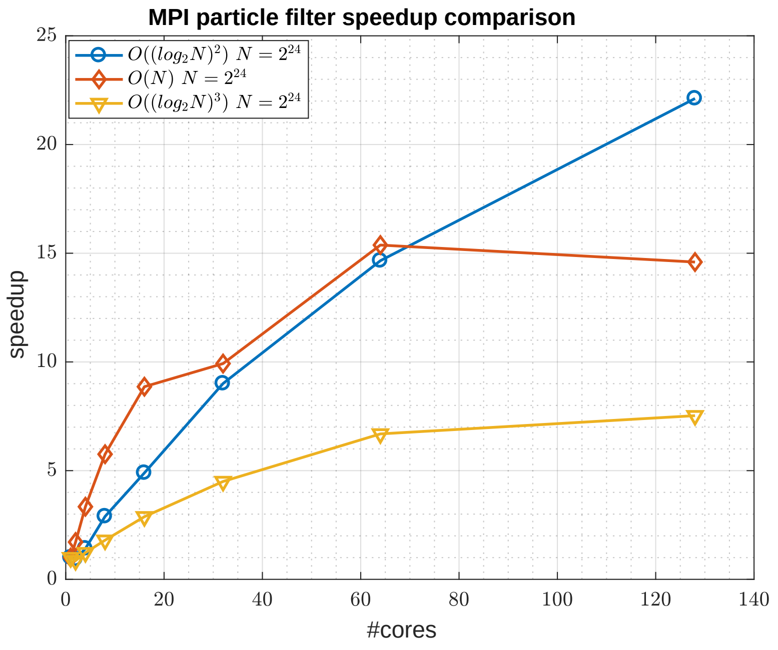


FIGURE A.10: Speedup: MPI particle filter ($N = 2^{24}$)

A.6 Conclusions and future work

In this appendix, we reformulated the particle filter algorithm outlined in [97] for the distributed memory setup. Our results suggest that a distributed implementation provides substantial speedups over basic and serial versions. Moreover, speedups and runtimes for the distributed memory setup are several times better than the results reported for the MapReduce platform in [97]. This enhancement is because the $O\left((\log_2 N)^2\right)$ particle filter is better than the $O\left((\log_2 N)^3\right)$ variant, and it can outperform the $O(N)$ variant for a relatively small number of cores, which does not occur on MapReduce.

The findings presented here are very encouraging. However, the overall particle filter algorithm, and the components therein can be improved in several ways. One key observation is that the current implementation includes the notion that all processors or cores are purely distributed. In practice, this scenario is not the case. Instead, cores are grouped inside a node with node-memory to provide some locality to the cores and become distributed in space. Such an architectural arrangement can be exploited by using the shared-memory parallelism within nodes and distributed memory parallelism across nodes. Another avenue of exploration is to guarantee the performance behaviours of sorting algorithms. For instance, the Serial Bitonic sort in Algorithm 16 can be replaced by a better single core sorting algorithm, such as Merge Sort. Another approach is to engineer these algorithms on novel and upcoming architectures, such as vector processors and FPGAs.

Table A.4: Details of the Experimental Platform.

OS	Linux
Number of Nodes	8
Cores per node	16
RAM	64GB
CPU Core	Xeon(R) CPU E5-2660
Clock	2.2GHz
L2	20MB
MPI Version	OpenMPI-1.5.3
Interconnect	Infiniband – 40Gbps

Appendix B

Variance of an importance sampler

While derivations of the unbiased nature of importance sampling and its variance exist, we provide an articulation of a step-by-step argument accessible to an engineering audience who might otherwise find the statistics somewhat unfamiliar.

B.1 Importance sampling estimator is unbiased

Assume we draw N samples of x from $q\left(x\right)$ such that the joint density of the N samples is $q\left(x^{1:N}\right)$. We show that $\hat{f}$ is an unbiased estimate of $\bar{f}$, where:

$$\bar{f} = \int f\left(x\right)\pi\left(x\right)dx \tag{B.1}$$

$$\hat{f} = \frac{1}{N}\sum_{i=1}^{N}\frac{\pi\left(x^{i}\right)}{q(x^{i})}f(x^{i}) \tag{B.2}$$

Consider an expectation of $\hat{f}$ over the density of the samples

$$\mathbb{E}_{q(x^{1:N})}\left[\hat{f}\right] = \mathbb{E}_{q(x^{1:N})}\left[\frac{1}{N}\sum_{i=1}^{N}\frac{\pi\left(x^i\right)}{q(x^i)}f(x^i)\right] \tag{B.3}$$

$$= \int\left(\frac{1}{N}\sum_{i=1}^{N}\frac{\pi\left(x^i\right)}{q(x^i)}f\left(x^i\right)\right)q(x^{1:N})dx^{1:N} \tag{B.4}$$

$$= \int\left(\frac{1}{N}\sum_{i=1}^{N}\frac{\pi\left(x^i\right)}{q(x^i)}f\left(x^i\right)\right)\prod_{i=1}^{N}q\left(x^i\right)dx^{1:N} \tag{B.5}$$

$$= \frac{1}{N}\sum_{i=1}^{N}\left(\underbrace{\int\frac{\pi(x^i)}{q(x^i)}f(x^i)q(x^i)dx^i}_{=\bar{f}}\times \underbrace{\left(\int\frac{\prod_{j=1}^{N}q(x^j)}{q(x^i)}dx^1\dots dx^{i-1}dx^{i+1}\dots dx^N\right)}_{=1}\right) \tag{B.6}$$

$$= \bar{f} \tag{B.7}$$

We can calculate the normalising constant for the joint density as

$$c_{1:k} = \int\pi_{1:k}\left(x_{1:k}\right)dx_{1:k} = \int\pi_{1:k}\left(x_{1:k}\right)\underbrace{\frac{q\left(x_{1:k}\right)}{q\left(x_{1:k}\right)}}_{=1}dx_{1:k} \tag{B.8}$$

$$\approx \sum_{i=1}^{N}\frac{\pi_{1:k}\left(x_{1:k}^{(i)}\right)}{q\left(x_{1:k}^{(i)}\right)} \tag{B.9}$$

$$= \sum_{i=1}^{N}w_k^{(i)} \tag{B.10}$$

which we note is the same as using (5.1) with $f\left(x\right) = 1$. Using (B.10), we can then define $\tilde{w}_k^{(i)}$, a normalised weight, as

$$\tilde{w}_k^{(i)} = \frac{w_k^{(i)}}{\sum_{j=1}^{N}w_k^{(j)}} \tag{B.11}$$

An estimate based on the normalised weights will, in general, be biased although with often lower variance than $\hat{f}_k$ such that

$$\tilde{f} = \sum_{i=1}^{N}\tilde{w}_k^{(i)}f\left(x_k^{(i)}\right) \neq \hat{f}_k \tag{B.12}$$

B.2 Variance

We use a similar approach to derive an expression for the variance, σ^2, of the estimate (i.e. the variance associated with $\hat{f}$) as

$$\sigma^2 = \mathbb{E}_{q(x^{1:N})}\left[\left(\bar{f}-\hat{f}\right)^2\right] \tag{B.13}$$

$$= \mathbb{E}_{q(x^{1:N})}\left[\bar{f}^2 - 2\hat{f}\bar{f} + \hat{f}^2\right] \tag{B.14}$$

$$= \bar{f}^2 - 2\mathbb{E}_{q(x^{1:N})}\left[\hat{f}\right]\bar{f} + \mathbb{E}_{q(x^{1:N})}\left[\hat{f}^2\right] \tag{B.15}$$

$$= \bar{f}^2 - 2\bar{f}^2 + \mathbb{E}_{q(x^{1:N})}\left[\hat{f}^2\right] \tag{B.16}$$

$$= \mathbb{E}_{q(x^{1:N})}\left[\hat{f}^2\right] - \bar{f}^2 \tag{B.17}$$

$$= \int \left(\frac{1}{N}\sum_{i=1}^{N}\frac{\pi\left(x^i\right)}{q(x^i)}f(x^i)\right)^2 q(x^{1:N})dx^{1:N} - \bar{f}^2 \tag{B.18}$$

$$= \int \left(\frac{1}{N}\sum_{i=1}^{N}\frac{\pi\left(x^i\right)}{q(x^i)}f(x^i)\right)^2 \prod_{i=1}^{N} q\left(x^i\right)dx^{1:N} - \bar{f}^2 \tag{B.19}$$

$$= \int \frac{1}{N}\left(\sum_{i=1,i\neq j}^{N}\sum_{j=1}^{N}\frac{\pi\left(x^i\right)}{q(x^i)}f(x^i)\frac{\pi\left(x^i\right)}{q(x^j)}f(x^j) + \sum_{i=1}^{N}\frac{\pi\left(x^i\right)^2}{q(x^i)^2}f(x^i)^2\right)$$
$$\prod_{i=1}^{N} q\left(x^i\right)dx^{1:N} - \bar{f}^2 \tag{B.20}$$

$$= \frac{1}{N^2}\left(\sum_{i=1,i\neq j}^{N}\sum_{j=1}^{N}\underbrace{\int\frac{\pi(x^i)}{q(x^i)}f(x^i)q(x^i)dx^i}_{=\bar{f}}\underbrace{\int\frac{\pi(x^j)}{q(x^j)}f(x^j)q(x^j)dx^j}_{=\bar{f}} + \right.$$
$$\left.\sum_{i=1}^{N}\int\frac{\pi(x^i)^2}{q(x^i)^2}f(x^i)^2q(x^i)dx^i\right) - \bar{f}^2 \tag{B.21}$$

$$= \frac{1}{N^2}\left(\left(N^2-N\right)\bar{f}^2 + \sum_{i=1}^{N}\int\frac{\pi\left(x^i\right)^2}{q(x^i)}f(x^i)^2dx^i\right) - \bar{f}^2 \tag{B.22}$$

$$= \frac{1}{N^2}\left(\sum_{i=1}^{N}\left[\int\frac{\pi\left(x^i\right)^2}{q(x^i)}f(x^i)^2dx^i - \bar{f}^2\right]\right) \tag{B.23}$$

This derivation illustrates that any freedom to define $\pi(x)$ and $q(x)$ can be used to reduce the variance, σ^2.

B.3 The need for heavy tails

Indeed, we can go further with this derivation by considering

$$\sigma^2 < \frac{1}{N^2} \sum_{i=1}^{N} \underbrace{\int c\pi\left(x^i\right) f(x^i)^2 dx^i}_{c\mathbb{E}[f(x)^2]} \tag{B.24}$$

$$= \frac{c}{N} \mathbb{E}\left[f\left(x\right)^2\right] \tag{B.25}$$

where we assume that $\frac{\pi(x)}{q(x)} > c$. This argument explains why the variance of the importance sampling estimator scales as $\frac{1}{N}$ for whatever dimensionality of the space x lives. However, the need to define c means that we need to ensure that, for all values of x it hold that $\frac{\pi(x)}{q(x)} > c$. Considering what happens as x tends to ∞, it is clear that for c to be finite, we need $\pi\left(x\right)$ to approach zero quicker than $q\left(x\right)$. This behaviour will only occur if $q\left(x\right)$ is heavier tailed than $\pi\left(x\right)$. Hence, the bound on the variance (ie the accuracy) of an estimate derived from using importance sampling will only get better with larger N if the proposal, $q\left(x\right)$, is heavier tailed than the target, $\pi\left(x\right)$.

Bibliography

[1] *A. Y. Ng*, https://www.coursera.org/learn/machine-learning, 2012, [Online; accessed 29-Jan.-2019].

[2] *Apache Hadoop*, http://hadoop.apache.org, 2016, [Online; accessed 19-Mar.-2018].

[3] *Apache Mahout*, http://mahout.apache.org, 2016, [Online; accessed 19-Mar.-2018].

[4] *Apache Pig and Latin*, http://pig.apache.org, 2016, [Online; accessed 19-Mar.-2018].

[5] *Apache Spark*, http://spark.apache.org, 2016, [Online; accessed 19-Mar.-2018].

[6] *Apache Storm*, http://storm.apache.org, 2016, [Online; accessed 19-Mar.-2018].

[7] *Particle Filter Repository*, https://github.com/particlefilter/mrpf, 2017, [Online; accessed 19-Mar.-2018].

[8] M. A. Carreira-Perpinan and G. E. Hinton, *On contrastive divergence learning*, 01 2005.

[9] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski, *A learning algorithm for Boltzmann machines*, Cognitive Science **9** (1985), no. 1, 147 – 169.

[10] C. Andrieu, N. de Freitas, and A. Doucet, *Robust Full Bayesian Learning for Radial Basis Networks*, Neural computation **13 10** (2001), 2359–407.

[11] G. Arampatzis, D. Wälchli, P. Angelikopoulos, S. Wu, and P. Koumoutsakos, *Langevin Diffusion Transitional Markov Chain Monte Carlo with an Application to Pharmacodynamics*, arXiv:1610.05660 (2016).

[12] M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, *A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking*, IEEE Transactions on Signal Processing **50** (2002), no. 2, 174–188.

[13] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, *A View*

of the Parallel Computing Landscape, Communications of the ACM **52** (2009), no. 10, 56–67.

[14] F. Bai and X. Hu, *Cloud MapReduce for particle filter-based data assimilation for wildfire spread simulation*, Simulation Series **45** (2013).

[15] A. S. Bashi, V. P. Jilkov, X. R. Li, and H. Chen, *Distributed implementations of particle filters*, Sixth International Conference of Information Fusion **2** (2003), 1164–1171.

[16] K. E. Batcher, *Sorting Networks and Their Applications*, Proceedings of the Spring Joint Computer Conference, AFIPS '68 (Spring), ACM, 1968, pp. 307–314.

[17] Y. Bengio, *Learning Deep Architectures for AI*, Found. Trends Mach. Learn. **2** (2009), no. 1, 1–127.

[18] J. Bergstra and Y. Bengio, *Random Search for Hyper-parameter Optimization*, J. Mach. Learn. Res. **13** (2012), 281–305.

[19] W. Betz, I. Papaioannou, and D. Straub, *Transitional Markov Chain Monte Carlo: Observations and Improvements*, Journal of Engineering Mechanics **142** (2016), no. 5, 04016016.

[20] G. E. Blelloch, *Prefix Sums and Their Applications*, (1990), no. CMU-CS-90-190.

[21] M. Bolic, P. M. Djuric, and S. Hong, *Resampling Algorithms and Architectures for Distributed particle Filters*, IEEE Transactions on Signal Processing **53** (2005), no. 7, 2442–2450.

[22] B. Casella, G. Roberts, and O. Stramer, *Stability of Partially Implicit Langevin Schemes and Their MCMC Variants*, Methodology and Computing in Applied Probability **13** (2011), no. 4, 835–854.

[23] T. Chen, E. B. Fox, and C. Guestrin, *Stochastic gradient Hamiltonian Monte Carlo*, 31st International Conference on Machine Learning, ICML 2014 **5** (2014), 3663–3676.

[24] J. Ching and Y. Chen, *Transitional Markov Chain Monte Carlo Method for Bayesian Model Updating, Model Class Selection, and Model Averaging*, Journal of Engineering Mechanics **133** (2007), no. 7, 816–832.

[25] D. Creal, *A Survey of Sequential Monte Carlo Methods for Economics and Finance*, Econometric Reviews **31** (2012), no. 3, 245–296.

[26] G. Dahl, H. Larochelle, and R. P. Adams, *Training Restricted Boltzmann Machines on Word Observations*, Proceedings of the 29th International Conference on Machine Learning (ICML-12) (2012), 679–686.

[27] A. S. Dalalyan, *Theoretical guarantees for approximate sampling from smooth and log-concave densities*, Journal of the Royal Statistical Society: Series B (Statistical Methodology) **79** (2017), no. 3, 651–676.

[28] J. Dean and S. Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, Communications of the ACM **51** (2008), no. 1, 107–113.

[29] P. Del Moral, A. Doucet, and A. Jasra, *Sequential Monte Carlo samplers*, Journal of the Royal Statistical Society: Series B (Statistical Methodology) **68** (2006), no. 3, 411–436.

[30] S. Duane, A. D. Kennedy, B. J. Pendleton, and D. Roweth, *Hybrid Monte Carlo*, Physics Letters B **195** (1987), no. 2, 216 – 222.

[31] A. Durmus, G. O. Roberts, G. Vilmart, and K. C. Zygalakis, *Fast Langevin based algorithm for MCMC in high dimensions*, Annals of Applied Probability **27** (2017), no. 4, 2195–2237.

[32] A. Eberle, *Error bounds for Metropolis–Hastings algorithms applied to perturbations of Gaussian measures in high dimensions*, The Annals of Applied Probability **24** (2014), no. 1, 337–377.

[33] A. Fischer and C. Igel, *An Introduction to Restricted Boltzmann Machines*, Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications: 17th Iberoamerican Congress, CIARP (2012), 14–36.

[34] J. Geweke and G. Durham, *Massively Parallel Sequential Monte Carlo for Bayesian Inference*, SSRN Electronic Journal (2011).

[35] M. Girolami and B. Calderhead, *Riemann manifold Langevin and Hamiltonian Monte Carlo methods*, Journal of the Royal Statistical Society: Series B (Statistical Methodology) **73** (2011), no. 2, 123–214.

[36] P. Gong, Y. O. Basciftci, and F. Ozguner, *A Parallel Resampling Algorithm for particle Filtering on Shared-Memory Architectures*, IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum (2012), 1477–1483.

[37] P. L. Green and S. Maskell, *Estimating the parameters of dynamical systems from Big Data using Sequential Monte Carlo samplers*, Mechanical Systems and Signal Processing **93** (2017), 379 – 396.

[38] P. L. Green and K. Worden, *Bayesian and Markov chain Monte Carlo methods for identifying nonlinear systems in the presence of uncertainty*, Philosophical Transactions Of The Royal Society A - Mathematical Physical And Engineering Sciences **373** (2015), no. 2051.

[39] W. G. Hatcher and W. Yu, *A survey of deep learning: Platforms, applications and emerging research trends*, IEEE Access **6** (2018), 24411–24432.

[40] S. Haykin, *Neural Networks: A Comprehensive Foundation*, Prentice Hall PTR (1998).

[41] G. Hendeby, R. Karlsson, and F. Gustafsson, *Particle Filtering: The Need for Speed*, EURASIP Journal of Advances in Signal Processing **2010** (2010), 1–9.

[42] C. Herath and B. Plale, *Streamflow Programming Model for Data Streaming in Scientific Workflows*, CCGRID, IEEE Computer Society, 2010, pp. 302–311.

[43] M. D. Hill, *What is Scalability?*, SIGARCH Comp. Arch. News **18** (1990), no. 4, 18–21.

[44] G. E. Hinton, *Boltzmann Machines*, Scholarpedia (2007).

[45] G. E. Hinton, S. Osindero, and Y. Teh, *A Fast Learning Algorithm for Deep Belief Nets*, Neural Comput. **18** (2006), no. 7, 1527–1554.

[46] C. A. R. Hoare, *Algorithm 64: Quicksort*, Communications of the ACM **4** (1961), no. 7, 321.

[47] J. D. Hol, T. B. Schon, and F. Gustafsson, *On Resampling Algorithms for particle Filters*, IEEE Nonlinear Statistical Signal Processing Workshop (2006), 79–82.

[48] K. Hornik, *Approximation capabilities of multilayer feedforward networks*, Neural Networks **4** (1991), no. 2, 251 – 257.

[49] K. Hwang and W. Sung, *Load Balanced Resampling for Real-Time particle Filtering on Graphics Processing Units*, Transactions in Signal Processing **61** (2013), no. 2, 411–419.

[50] K. Ito, D. W. Stroock, and S. R. S. Varadhan, *Kiyosi Ito selected papers / edited by Daniel W. Stroock, S. R. S. Varadhan*, SERBIULA (sistema Librum 2.0) (2017).

[51] K. E. Iverson, *A Programming Language*, John Wiley & Sons, Inc., New York, NY, USA, 1962.

[52] V. P. Jilkov and J. Wu, *Implementation and performance of a parallel multitarget tracking particle filter*, 14th International Conference on Information Fusion, July 2011, pp. 1–8.

[53] Sharon Khan and Andy M. Reynolds, *Derivation of a fokker–planck equation for generalized langevin dynamics*, Physica A: Statistical Mechanics and its Applications **350** (2005), no. 2, 183 – 188.

[54] J. H. Kotecha and P. M. Djuric, *Gaussian particle filtering*, IEEE Transactions on Signal Processing **51** (2003), no. 10, 2592–2601.

[55] Y. Lecun, Y. Bengio, and G. Hinton, *Deep learning*, Nature (2015), no. 7553.

[56] T. Li, M. Bolic, and P. M. Djuric, *Resampling Methods for particle Filtering: Classification, implementation, and strategies*, IEEE Signal Processing Magazine **32** (2015), no. 3, 70–86.

[57] G. Litjens, T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. W. M. van der Laak, B. van Ginneken, and C. I. Sánchez, *A survey on deep learning in medical image analysis*, Medical Image Analysis **42** (2017), 60 – 88.

[58] D. C. Liu and J. Nocedal, *On the limited memory BFGS method for large scale optimization*, Mathematical Programming **45** (1989), no. 1, 503–528.

[59] J. S. Liu, *Metropolized independent sampling with comparisons to rejection sampling and importance sampling*, Statistics and Computing **6** (1996), no. 2, 113–119.

[60] J. S. Liu and R. Chen, *Sequential Monte Carlo Methods for Dynamic Systems*, Journal of the American Statistical Association **93** (1998), no. 443, 1032–1044.

[61] K. Liu, L. Tang, S. Li, L. Wang, and W. Liu, *Parallel particle filter algorithm in face tracking*, IEEE International Conference on Multimedia and Expo (2009), 1817–1820.

[62] S. Liu, G. Mingas, and C. S. Bouganis, *Parallel resampling for particle filters on FPGAs*, International Conference on Field-Programmable Technology (FPT) (2014), 191–198.

[63] F. Lopez, L. Zhang, A. Mok, and J. Beaman, *particle filtering on GPU architectures for manufacturing applications*, Computers in Industry **71** (2015), 116 – 127.

[64] T. Lux, *Estimation of agent-based models using sequential Monte Carlo methods*, Journal of Economic Dynamics and Control (2017), no. 2017-07.

[65] T. Marshall and G. Roberts, *An Adaptive Approach to Langevin MCMC*, Statistics and Computing **22** (2012), no. 5, 1041–1057.

[66] S. Maskell, *An application of Sequential Monte Carlo samplers: An alternative to particle filters for non-linear non-Gaussian sequential inference with zero process noise*, 9th IET Data Fusion Target Tracking Conference: Algorithms Applications (2012), 1–8.

[67] S. Maskell, B. Alun-Jones, and M. Macleod, *A Single Instruction Multiple Data particle Filter*, IEEE Nonlinear Statistical Signal Processing Workshop (2006), 51–54.

[68] S. Maskell and S. Julier, *Optimised proposals for improved propagation of multimodal distributions in particle filters*, Proceedings of the 16th International Conference on Information Fusion (2013), 296–303.

[69] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, *Equation of State Calculations by Fast Computing Machines*, The Journal of Chemical Physics **21** (1953), no. 6, 1087–1092.

[70] L. Miao, J. J. Zhang, C. Chakrabarti, and A. Papandreou-Suppappola, *A new parallel implementation for particle filters and its application to adaptive waveform design*, IEEE Workshop On Signal Processing Systems (2010), 19–24.

[71] A. Mnih and K. Gregor, *Neural Variational Inference and Learning in Belief Networks*, Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32 (2014), II–1791–II–1799.

[72] P. Del Moral, A. Doucet, and A. Jasra, *Sequential Monte Carlo samplers*, Journal of the Royal Statistical Society: Series B (Statistical Methodology) **68** (2006), no. 3, 411–436.

[73] L. M. Murray, A. Lee, and P. E. Jacob, *Parallel resampling in the particle filter*, Journal of Computational and Graphical Statistics **25** (2016), no. 3, 789–805.

[74] V. Nair and G. E. Hinton, *3D Object Recognition with Deep Belief Nets*, Advances in Neural Information Processing Systems 22 (Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta, eds.), Curran Associates, Inc., 2009, pp. 1339–1347.

[75] R. M. Neal, *Connectionist Learning of Belief Networks*, Artif. Intell. **56** (1992), no. 1, 71–113.

[76] R. M. Neal, *MCMC Using Hamiltonian Dynamics*, Handbook of Markov Chain Monte Carlo **54** (2010), no. 3, 113–162.

[77] A. Ng, J. Ngiam, C. Y. Foo, Y. Mai, and C. Suen, *UFLDL Tutorial*, (2010).

[78] T. L. T. Nguyen, F. Septier, G. W. Peters, and Y. Delignon, *Efficient Sequential Monte-Carlo Samplers for Bayesian Inference*, IEEE Transactions on Signal Processing **64** (2016), no. 5, 1305–1319.

[79] T. Ozaki, *A bridge between nonlinear time series models and nonlinear stochastic dynamical systems: a local linearization approach*, Statistica Sinica (1992), 113–135.

[80] G. Panis and A. Lanitis, *An Overview of Research Activities in Facial Age Estimation Using the FG-NET Aging Database*, Computer Vision - ECCV 2014 Workshops: Zurich, Switzerland, September 6-7 and 12, 2014, Proceedings, Part II (2014), 737–750.

[81] G. Pavliotis, *Stochastic Processes and Applications: Diffusion Processes, the Fokker-Planck and Langevin Equations*, (2014), 87–137.

[82] J. E. Pow, N. and L. Liu, *Applied Machine Learning Project for Prediction of real estate property prices in Montreal*, (2014).

[83] Matthew T. Pratola, *Efficient metropolis–hastings proposal mechanisms for bayesian regression tree models*, Bayesian Anal. **11** (2016), no. 3, 885–911.

[84] G. R. Price, *Extension of covariance selection mathematics*, Annals of human genetics **35** (1972), no. 4, 485–490.

[85] A. A. A. Rahni, E. Lewis, M. J. Guy, B. Goswami, and K. Wells, *Performance evaluation of a particle filter framework for respiratory motion estimation in Nuclear Medicine imaging*, IEEE Nuclear Science Symposium Medical Imaging Conference (2010), 2676–2680.

[86] J. L. Reyes-Ortiz, L. Oneto, and D. Anguita, *Big Data Analytics in the Cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf*, Procedia Computer Science **53** (2015), 121–130.

[87] G. O. Roberts and J. S. Rosenthal, *Complexity bounds for Markov chain Monte Carlo algorithms via diffusion limits*, Journal of Applied Probability **53** (2016), no. 2, 410–420.

[88] Gareth O. Roberts and Richard L. Tweedie, *Exponential convergence of langevin distributions and their discrete approximations*, Bernoulli **2** (1996), no. 4, 341–363.

[89] Sebastian Ruder, *An overview of gradient descent optimization algorithms.*, CoRR **abs/1609.04747** (2016).

[90] M. Schroeck, R. Shockley, J. Smart, D. Romero-Morales, and P. Tufano, *Analytics: The Real-world Use of Big data*, IBM Institute for Business Value, IBM Institute for Business Value - Executive Report (2012).

[91] F. Schwiegelshohn, E. Ossovski, and M. Hübner, *A Fully Parallel particle Filter Architecture for FPGAs*, pp. 91–102, Springer, 2015.

[92] M. Shabany and P. G. Gulak, *An efficient architecture for distributed resampling for high-speed particle filtering*, 2006 IEEE International Symposium on Circuits and Systems (2006), 4 pp.–3425.

[93] G. S. Simões, J. Wehrmann, R. C. Barros, and D. D. Ruiz, *Movie genre classification with Convolutional Neural Networks*, International Joint Conference on Neural Networks (IJCNN) (2016), 259–266.

[94] D. Singh and C. K. Reddy, *A survey on platforms for big data analytics*, Journal of Big Data **2** (2014), no. 1, 1–20.

[95] S. Sutharsan, T. Kirubarajan, T. Lang, and M. McDonald, *An optimization-based parallel particle filter for multitarget tracking*, IEEE Transactions on Aerospace and Electronic Systems **48** (2012), no. 2, 1601–1618, cited By 25.

[96] A. Tahara, Y. Hayashida, T. T. Thu, Y. Shibata, and K. Oguri, *FPGA-based Real-Time Object Tracking Using a particle Filter with Stream Architecture*, 2016 Fourth International Symposium on Computing and Networking (CANDAR) (2016), 422–428.

[97] J. Thiyagalingam, L. Kekempanos, and S. Maskell, *MapReduce particle filtering with exact resampling and deterministic runtime*, EURASIP Journal on Advances in Signal Processing **2017** (2017), no. 1, 71.

[98] A. H. Thiéry, A. M. Stuart, and N. S. Pillai, *Optimal scaling and diffusion limits for the Langevin algorithm in high dimensions*, The Annals of Applied Probability **22** (2012), no. 6, 2320–2356.

[99] S. Thrun, *Particle Filters in Robotics*, Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence, UAI'02, 2002, pp. 511–518.

[100] Don van Ravenzwaaij, Pete Cassey, and Scott D. Brown, *A simple introduction to Markov Chain Monte–Carlo sampling*, Psychonomic Bulletin & Review **25** (2018), no. 1, 143–154.

[101] A. Varsi, L. Kekempanos, J. Thiyagalingam, and S. Maskell, *Parallelising particle Filtering for Deterministic Runtimes on Distributed Memory Systems*, IET 3rd International Conference on Intelligent Signal Processing (ISP) (2017).

[102] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P. Manzagol, *Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion*, J. Mach. Learn. Res. **11** (2010), 3371–3408.

[103] H. Wang, X. Qin, X. Zhou, F. Li, Z. Qin, Q. Zhu, and S. Wang, *Efficient Query Processing Framework for Big data Warehouse: An Almost Join-Free Approach*, Frontiers of Computer Science **9** (2015), no. 2, 224–236.

[104] M. Welling and Y. Teh, *Bayesian Learning via Stochastic Gradient Langevin Dynamics*, (2011), 681–688.

[105] Y. Wu, J. Wang, and Y. Cao, *Particle filter based on iterated importance density function and parallel resampling*, Journal of Central South University **22** (2015), no. 9, 3427–3439.

[106] J. Yosinski and H. Lipson, *Visually Debugging Restricted Boltzmann Machine Training with a 3D Example*, (2012).

[107] T. Young, D. Hazarika, S. Poria, and E. Cambria, *Recent Trends in Deep Learning Based Natural Language Processing*, CoRR **abs/1708.02709** (2017).

[108] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, *Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing*, Ninth USENIX Symposium on Networked Systems Design and Implementation (NSDI 12) (San Jose, CA), USENIX, 2012, pp. 15–28.

[109] J. Zhu, J. Chen, W. Hu, and B. Zhang, *Big Learning with Bayesian methods*, National Science Review **4** (2017), no. 4, 627–651.

[110] R. Zhu, Y. Long, Y. Zeng, and W. An, *Parallel particle PHD filter implemented on multicore and cluster systems*, Signal Processing **127** (2016), 206 – 216.

[111] R. Zwanzig, *Nonequilibrium statistical mechanics* , (2001) (English), Includes index.