UNIVERSITY OF LIVERPOOL

Streaming Multi-core Sample-based Bayesian Analysis

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

Alessandro Varsi

June 2021

To my grandparents

Contents

Illustrations

List of Figures

List of Tables

Abbreviations

The following abbreviations are found throughout this thesis:

API	Application Programming Interface
B-R	Bitonic sort based Redistribute
C-R	Centralised Redistribute
CPU	Computing Processing Unit
CDF	Cumulative Density Function
DOP	Degree Of Parallelism
DMA	Distributed Memory Architecture
ESS	Effective Sample Size
FPGA	Field Programmable Gate Array
FL-HMC	Fixed-Lag Hamiltonian Monte Carlo
FL-NUTS	Fixed-Lag No-U-Turn Sampler
FL-SMC	Fixed-Lag Sequential Monte Carlo
GPU	Graphics Processing Unit
HMC	Hamiltonian Monte Carlo
HPC	High Performance Computing
HMA	Hybrid Memory Architecture
IS	Importance Sampling
LSB	Least Significant Bit
MCMC	Markov Chain Monte Carlo
MPI	Message Passing Interface
MVR	Minimum Variance Resampling
MC	Monte Carlo
MSB	Most Significant Bit
N-R	Nearly sort based Redistribute
NUTS	No-U-Turn Sampler
PDE	Partial Differential Equation
PF	Particle Filter
PDF	Probability Density Function
RNG	Random Number Generator
RW	Random Walk

RNA	Resampling with Non-proportional Allocation
RPA	Resampling with Proportional Allocation
RMSE	Root Mean Squared Error
RoSS	Rotational nearly Sort and Split
SIR	Sequential Importance Resampling
SMC-S	Sequential Importance Sampling
SMCMC	Sequential Markov Chain Monte Carlo
SMC	Sequential Monte Carlo
S-R	Sequential Redistribute
SMA	Shared Memory Architecture

Abstract

Sequential Monte Carlo (SMC) methods are a well-established family of Bayesian inference algorithms for performing state estimation for Non-Linear Non-Gaussian models. As the models become more accurate, the run-time of SMC applications becomes increasingly slow. Parallel computing can be used to compensate for this side-effect. However, an efficient parallelisation of SMC is hard to achieve, due to the challenges involved in parallelising the bottleneck, resampling, and its constituent redistribute step. While redistribution can be performed in $O(\frac{N}{T}\log_2 N)$ on a Shared Memory Architecture (SMA) using T parallel threads (e.g. a GPU or mainstream CPUs), a state-of-the-art redistribute takes $O((\log_2 N)^2)$ computations on Distributed Memory Architectures (DMAs) which most supercomputers are made of. In this thesis, the focus is on three major goals.

First, the thesis proposes a novel parallel redistribute for DMAs which achieves $O(\log_2 N)$ time complexity. It is shown that on Message Passing Interface (MPI) the novel redistribute is up to eight times faster than the $O((\log_2 N)^2)$ one. On a cluster of 256 cores, an SMC method employing the $O((\log_2 N)^2)$ redistribute becomes up to six times faster when switching to the novel redistribution, which is also proved to no longer be the bottleneck. For the same number of cores, the maximum reported speed-up vs a single-core SMC method is 160. A patent application on this invention is currently filed.

Second, the thesis describes a novel parallel redistribute for SMAs which takes $O(\frac{N}{T} + \log_2 N)$ steps and fully exploits the computational power of SMAs. The proposed approach is up to six times faster than the $O(\frac{N}{T}\log_2 N)$ one. This shared memory implementation is then combined with the MPI $O(\log_2 N)$ redistribution to obtain a hybrid distributed-shared memory parallel redistribute that fully exploits the large parallelism that modern supercomputers offer.

In the end, to make these advances widely available this thesis presents Streaming-Stan and SMC-Stan, two extension packages for Stan, a popular statistical programming language. Streaming-Stan and SMC-Stan offer the possibility to describe models by using the same intuitive syntax used by regular Stan, but they are also equipped with the aforementioned High Performance Computing (HPC) SMC method, in the form of Fixed-Lag SMC and SMC sampler respectively. The same SMC methods also provide a vast choice of proposal distributions, including (on Streaming-Stan) two novel ones, presented in this thesis, which combine the main features of Fixed-Lag SMC methods with Hamiltonian Monte Carlo (HMC) and No-U-Turn Sampler (NUTS).

Acknowledgements

I am profoundly grateful to my primary supervisor, Professor Simon Maskell, for giving me the opportunity of being one of his PhD students, for being a great mentor, and for assigning me this challenging problem that matches so well my research interests.

I thank Dr Jeyarajan Thyiagalingam for teaching me many parallel computing tricks that cannot be found in textbooks, during the two years he was my secondary supervisor.

I thank my advisors Dr Peter Green and Dr Simon Maher for their suggestions regarding my thesis during our annual interviews.

I gratefully acknowledge my PhD sponsors: the UK EPSRC Doctoral Training Award and Schlumberger for their support. I also gratefully acknowledge IBM Research UK for giving me the opportunity of doing an internship at their facility in Daresbury, UK. In particular, I thank Dr Inês Cecilio from Schlumberger and Dr Edward-Pyzer Knapp from IBM for the advice that they have provided.

I thank Dr Lykourgos Kekempanos for all the interesting discussions about this project. I also thank my other lab-mates: Robert Moore, Dr Lee Devlin, Elpida Kontsioti, Dr Philip Clemson, Dr Lyudmil Vladimirov, Dr Alexander Phillips, Dr James Write, Dr Roberta Piroddi, Katerina Chatzopoulou, Dr Paul Horridge, Dr Yifan Zhou, Josh Coates and Dr Chinmay Mishra.

I thank all my Italian friends in UK and back in Italy: Maurizio, Dr Andrea Picciau, Enrica, Andrea Ambu, Daniele Lai, Davide, Sara, Mauro, Ilenia, Fabrizio, Stefania, Roberto, Veronica, Gianluca, Dr Ilaria Frau, Andrea Mulas, Valentina, Francesco, Dr Gabriele Irde, Giacomo, Alessia, Luca, Susanna, Riccardo Saliu, Riccardo Zuddas, Daniele Dentoni, Dr Valerio Selis and Dr Alessandra Frau.

I thank my relatives and those friends I have always considered relatives. You are too many, and I only have one page for this, but you know that I am talking about you. But a special thank goes to my uncles Antonello, Giuseppe and Emanuele.

I thank my sister Roberta for always being a model of academic excellence.

I thank Alyson for her constant support during these four years.

Last but not least, I thank my parents Patrizia and Antonio for so many things: for being my first teachers, for their advice, support and for sacrificing so much of what they could have been so that my sister and I could have a high level education.

Grazie mille a tutti.

Gratzias meda a totus.

Chapter 1

Introduction

Future events are and will always be an object of concern. Common daily questions could be about, for example, tomorrow's weather, stock markets or, unfortunately, reported cases during a pandemic. The inherent random nature of future events indeed influences many aspects of our present, especially the decisions we make and the margin of risk that comes with them. Therefore, making accurate predictions is a subject of great interest. However, since time is also a major contributing factor to the efficacy and the effectiveness of our decisions, making predictions both accurately and as quickly as possible is usually critical. These requests often arise in a wide variety of real-world problems where a considerable amount of money or lives are on the line. It is then crucially important to provide the world with a tool to address these problems.

1.1 Overview

There exist several methodologies to perform statistical inference. Bayesian inference has been one of the most commonly used in the past 70 years. This approach is based on the well-know Bayes' theorem which states that a good knowledge of the past based on our personal experience, called *prior*, wisely combined with actual observations of present events, called *data*, give you a posteriori probability distribution, named *posterior* or *target*, from which we can make fairly good estimates of the future. Predictions also become more accurate as new data is collected, making Bayesian inference a versatile methodology in the context of both on-line and off-line applications. It is then not surprising that Bayesian inference methods find application in a wide variety of domains, ranging from economics to medicine, meteorology or, broadly speaking, any field where it is important to collect data and make predictions afterwards. For example, one could be interested in estimating the trajectory of a hurricane, given data of wind speed, temperature and atmospheric pressure; another example could be predictions of reported COVID-19 positive cases within a few weeks, after new restrictions are introduced.

In other words, for any system we are given its *model*, a set of mathematical equations designed to describe the *state*, a collection of the most characteristic physical quantities of the system. Since models usually describe states only under ideal scenarios, i.e.

assuming no disturbance from the external environment occurs, traditional numerical methods do not always guarantee accurate results. We wish to apply Bayesian inference methods to convey uncertainty in the state estimation process. Bayesian inference is itself a large family of algorithms, which are classified within sub-classes, each of them providing advantages and disadvantages.

Sequential Monte Carlo (SMC) methods are statistical algorithms which are commonly employed to make Bayesian inferences in the context of either *dynamic models* (whose posterior is time-variant) or *static models* (whose target is constant in time). The application domain is therefore vast and diverse: in the literature we can find applications in positioning [35], medical research [22, 24], risk analysis [97], weather forecasting [88] or financial econometrics [51]. Particle Filters (PFs) [6] and SMC samplers [61] are two well-known examples of SMC methods. The first is broadly used for dynamic models under non-liner and non-Gaussian scenarios, while SMC samplers are more frequently applied to static models.

The key idea behind all SMC methods is to generate a population of N hypotheses of the true state, called *particles* or *samples*, by randomly selecting them from a user-defined probability distribution, called *proposal*. The particles are then confronted with the data, i.e. measurable quantities related to the state. This way we can assign to each sample a weight which functions as a probability of how well that particle resembles the true state. At any given iteration, the proposed estimate will be the weighted mean of the particles. These steps are then repeated iteratively until the simulation is stopped.

Since the trend is to make the models more and more detailed and complex, modern applications of SMC methods naturally get increasingly demanding in terms of accuracy constraints. Therefore, the research community is highly active on investigating solutions to satisfy this request. It is widely known that accuracy can trivially be improved by using more particles. That is because the proposal is proven to converge to the true distribution of interest as N increases [6]. This approach is often used as long as enough memory resources are available [52, 53]. However, it is also possible to improve the accuracy of the estimate by using several alternative strategies, ranging from employing a more sophisticated proposal distribution [63] to collecting more measurements if possible [99]. Applying any of these solutions, however, is likely to significantly slow down the run-time which could also become even more problematic if the constraint on the measurement rate is strict. In order to compensate for this side-effect without losing accuracy (e.g. by avoiding decreasing the number of particles), SMC methods need to use parallel computing, a type of computing technique which employs P processing elements, called *cores*, to enhance the run-time of an algorithm. The goal is therefore to provide *speed-up*, which is straightforwardly defined as the solution time of an algorithm run by one core divided by the solution time of the same algorithm run by $P > 1$ cores. More precisely, the focus is on *strong scaling* performance which defines how the run-time varies as a function of P for a fixed total problem size.

1.2 Motivations

SMC methods are often claimed to be trivially parallelisable because the samples are statistically independent of each other, meaning that they can be generated and weighted in embarrassingly parallel fashion. Although this statement is correct, at some point it becomes necessary to perform a *resampling* step in order to correct the weight *degeneration* of the particles, a problem which is inevitably caused by the sampling technique [6]. A textbook implementation of this resampling step is impossible to parallelise by using standard embarrassingly parallel computing techniques. Previous research has however demonstrated that it is possible to parallelise the resampling operation by using a divide-and-conquer strategy [58, 85]. The achieved time complexity is $O((\log_2 N)^2)$ for $P = N$ parallel cores. However, implementations of this solution on several frameworks, such as Hadoop or Spark, have shown little to no scalability, even for a large number of cores (see Figure 1.1). It is therefore necessary to investigate alternative scalable solutions for resampling to meet the accuracy constraints by fully exploiting the increasing computational power of modern computers and supercomputers (see Figure 1.2).

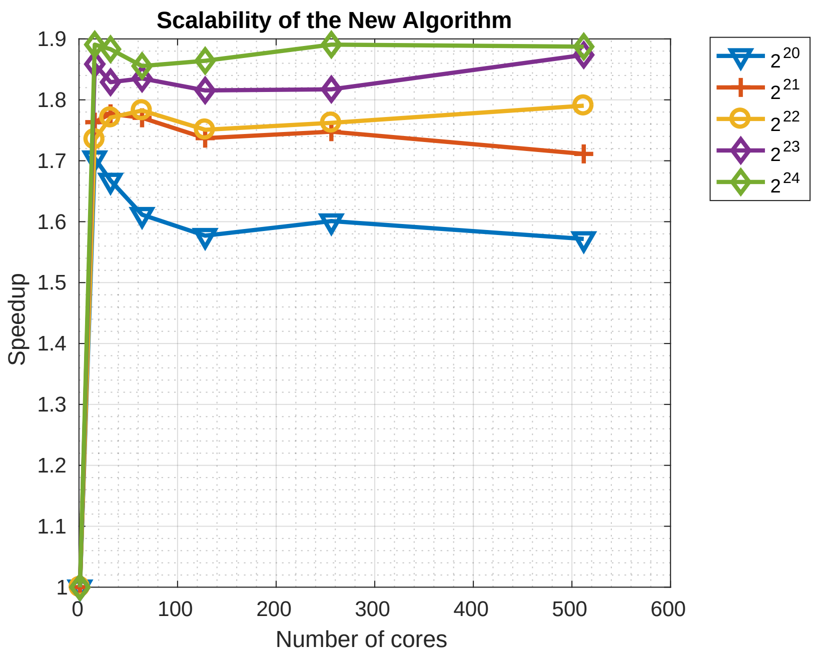


FIGURE 1.1: State-of-art: speed-up of the resampling parallelisation in [85] for increasing N.

In doing so, several questions need to be answered. The first is to identify a more suitable framework than MapReduce (as used in [85]) for divide-and-conquer resampling parallelisations. The second is to develop alternative parallel solutions for resampling, hopefully achieving a faster time complexity. It is also necessary to conduct a thorough investigation of the achieved performance of the resulting SMC method under various conditions, such as different computer architectures, increasing N or number of cores, different models or proposal distributions. Although SMC methods are widely popular, the vast majority of users mostly employ them as black-box, and even fewer have parallel computing background. Therefore, it would also be greatly helpful to provide a probabilistic programming language, with a succinct, user-friendly syntax and, most importantly, which builds on an optimised parallel implementation of an SMC method. This would make the improvements on SMC methods in the High Performance Computing (HPC) context easily accessible to anyone, with negligible learning overhead.

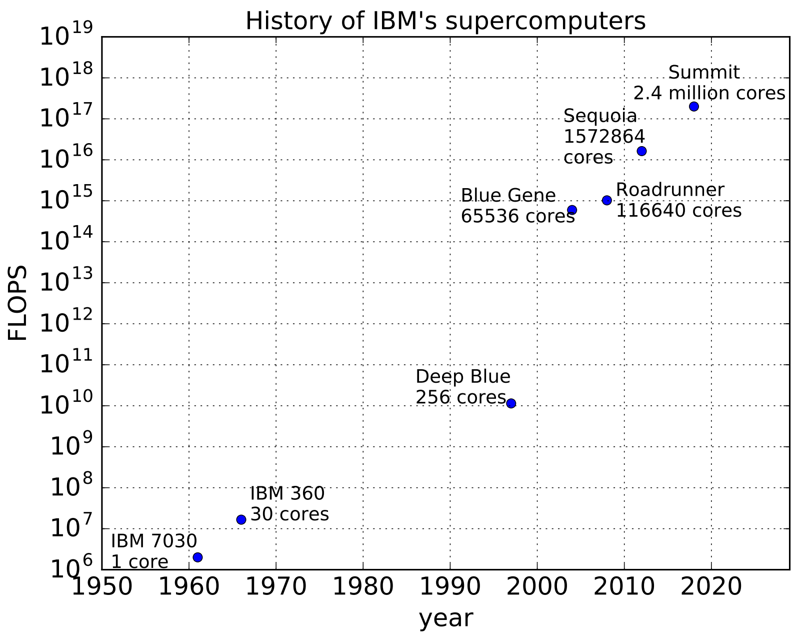


FIGURE 1.2: History of IBM's supercomputers

1.3 Contributions

The first major contribution of this thesis is to propose Message Passing Interface (MPI), a popular parallel programming model for distributed memory systems, as the ideal framework to implement the resampling parallelisation in [85]. Along with its implementation on MPI, further optimisations are also presented. For this contribution, my role has been to port the algorithm from MapReduce to MPI and gather the numerical results on a supercomputer. Precious tips during the implementation phase have been offered by Dr Lykourgos Kekempanos, Dr Jeyarajan Thiyagalingam and Professor Simon Maskell as authors of reference [85]. This contribution is described in Chapter 3.

The second major contribution is to fully redesign the same algorithm and propose a novel one on MPI that achieves $O(\log_2 N)$ time complexity. This algorithm has been co-invented by me and Professor Simon Maskell. I also entirely coded the algorithm and collected the experimental results. This invention is presented in Chapter 4.

The third contribution is to design shared memory parallel solutions for each single-node task of the novel MPI $O(\log_2 N)$ resampling; the preferred shared memory Application Programming Interface (API) is OpenMP. This way it is possible to embed the OpenMP components within the novel MPI resampling and, therefore, obtain a hybrid distributed-shared memory version of that which fully exploits the computational resources that modern supercomputers offer. These shared memory parallel solutions have been co-designed by me and Professor Simon Maskell. The coding and experimental phases were conducted by me but useful coding tips during were gently given by Jack Taylor, and Professor Vassil Alexandrov. This contribution is described in Chapter 5.

The final contribution is to develop two extension packages for Stan, a probabilistic programming language which performs Bayesian inference on static models by using a sampling algorithm called No-U-Turn Sampler (NUTS), as often used in the context of Markov Chain Monte Carlo (MCMC) methods. Stan currently counts about 10 thousand users, mostly statisticians, and its success is due to its intuitive, succinct syntax and its ability to interface with popular programming languages such as Python, MATLAB and R. These extensions have preserved all functionalities and syntax of Stan but they also embodies a PF or an SMC sampler, depending on the package. Since Stan has a back end written in C++, it is possible to embed an MPI+OpenMP parallel SMC method, having the same resampling parallelisation described in the previous contributions. Therefore, these extension packages can be installed and run on a modern supercomputer and can be applied in the context of either dynamic or static models. In each version, the user is also given the option of choosing from several proposal distributions, including two novel ones, which are co-invented by me and Professor Simon Maskell, and inspired from previous work done by Dr Paul Horridge and illustrated in Appendix D. I and the people working in my research group have agreed on naming these extension packages Streaming-Stan, in the case of PFs, and SMC-Stan, in the case of SMC samplers. The implementation of these packages (described in Appendix C of this thesis) has been mostly conducted by me but Robert Moore, Dr Lee Devlin, Dr Philip Clemson and Dr Alexander Phillips have been essential in finding some elusive bugs. The development of Streaming-Stan and SMC-Stan is the focal point of a research project, called Big Hypotheses, led by Professor Simon Maskell and closely coupled to work sponsored by many industrial partners such as Schlumberger (my PhD sponsor), IBM, Dstl, GCHQ, the National Crime Agency, AWE, MBDA and Leonardo. The novelty of this contribution is presented in Chapter 6.

1.4 Publications, Patents and Technical Work

The contributions described in Section 1.3 have produced the following published papers:

1. **A. Varsi**, L. Kekempanos, J. Thiyagalingam, and S. Maskell, “Parallelising Particle Filters with Deterministic Runtime on Distributed Memory Systems,” IET Conference Proceedings, pp. 11–18, 2017 (reference [89]). This paper is related to

the first major research outcome described in the previous section, and, therefore, my contribution has been at least equal to the other co-authors.

2. **A. Varsi**, J. Taylor, L. Kekempanos, E. Pyzer Knapp and S. Maskell, "A Fast Parallel Particle Filter for Shared Memory Systems," in IEEE Signal Processing Letters, vol. 27, pp. 1570-1574, 2020 (reference [91]). This paper presents the third major research outcome described in Section 1.3, to which my contribution has been at least equal to the other co-authors.

and the following papers, either submitted or still in preparation:

- **Alessandro Varsi** and Simon Maskell, "An $O(\log_2 N)$ Fully-Balanced Particle Filter for Distributed Memory Architectures", currently submitted to IEEE Transactions on Signal Processing. This paper is related to the second major research outcome described in Section 1.3, and my contribution to it has been at least equal to the other co-author.

- **A. Varsi**, L. Devlin, S. Maskell, "A Fixed-Lag SMC method with Hamiltonian moves for Long-Term Memory Models," which will be submitted to either IEEE Transactions on Signal Processing or IEEE Signal Processing Letters and is related to the fourth major contribution described in Section 1.3.

The same contributions have also produced the following filed patent:

- **Alessandro Varsi** and Simon Maskell, "Method Of Parallel Implementation in Distributed Memory Architectures". GB Patent Request 2101274.5, 29 Jan 2021 (reference [92]). This title has been made uninformative on purpose for Intellectual Property reasons. However, if the patent is granted, the title will be changed to: "An $O(\log_2 N)$ Fully-Balanced Redistribute for Sequential Monte Carlo methods on Distributed Memory Architectures" as this patent is related to the second major contribution discussed in Section 1.3.

Other technical work related to and cited in this thesis can be found in:

- **A. Varsi**, L. Kekempanos, J. Thiyagalingam, and S. Maskell, "A Single SMC Sampler on MPI that Outperforms a Single MCMC Sampler", eprint arXiv:1905.10252, 2019 (reference [90]).

1.5 Outline

The rest of the thesis is divided into six chapters and two appendixes.

1.5.1 Chapter 2 - Technical Background

Chapter 2 explains in details the necessary technical background to understand the chapters that follow. An explanation of the most important SMC methods is provided in Section 2.1, while details of the most useful MCMC methods to this thesis are given in Section 2.2.

1.5.2 Chapter 3 - Parallelising Particle Filters with Deterministic Run-time on Distributed Memory Systems

Chapter 3 starts with a thorough literature review and description of the most recent parallelisation strategies for resampling on distributed memory environments. Section 3.4 explains how to implement on MPI each of the key components of the resampling in [85]. In Section 3.4.5, the resulting algorithm is compared to another existing MPI implementation of resampling. Section 3.5 explains how to further optimise the same MPI resampling algorithm and Section 3.5.6 provides the results of the improvements and their impact on a toy problem solved by a PF.

1.5.3 Chapter 4 - An $O(\log_2 N)$ Fully-Balanced Particle Filter for Distributed Memory Architectures

In Chapter 4, the reader will see that, starting from the outcomes of Chapter 3, the invention of the novel $O(\log_2 N)$ parallel resampling on MPI is the result of three ideas described in Sections 4.2.1, 4.2.2 and 4.2.3. The first implementation of the $O(\log_2 N)$ resampling is presented in Section 4.3 and a further improvement of the same is described in Section 4.4. Sections 4.3.5 and 4.4.5 compare the novel MPI $O(\log_2 N)$ resampling with the algorithms presented in Chapter 3. In the same sections, the impact of the improvements are also studied on a PF working on two models, a toy problem and a real-world one.

1.5.4 Chapter 5 - A Fast Parallel Particle Filter on Hybrid Memory Architectures

Chapter 5 dedicates a section to each of the key components of the MPI $O(\log_2 N)$ SMC method from Chapter 4, in order to present for each of them a parallelisation on OpenMP. In particular, Section 5.2.4 describes a novel resampling parallelisation algorithm for SMAs that achieves $O(\log N)$ time complexity. In Section 5.3, an MPI+ OpenMP PF is present and compared to its MPI-only equivalent.

1.5.5 Chapter 6 - Streaming-Stan and SMC-Stan: Two High Performance Computing Extension Packages for Stan

Chapter 6 presents Streaming-Stan and SMC-Stan and describes how to use them. More precisely, Section 6.2 focuses on SMC-Stan and shows the numerical results on an exemplary SMC-Sampler for different proposal distributions. Section 6.3 illustrates Streaming-Stan and its proposal distributions, including two novel ones which combine the main features of Fixed-Lag SMC methods with Hamiltonian Monte Carlo (HMC) and NUTS. The performance of Streaming-Stan are demonstrated in Section 6.4 on multiple dynamic models and for several comparison metrics: flexibility, accuracy and run-time.

1.5.6 Chapter 7 - Conclusions

Chapter 7 highlights and summarises the most important outcomes of the research findings explained in the previous chapters. It also discusses ideas for future work, including those currently under development and those which are only planned for now.

1.5.7 Appendix A - Distributed, Shared and Hybrid Memory Architectures

Appendix A briefly describes distributed, shared and hybrid memory architectures and highlights their most important advantages and disadvantages. This appendix also provides a brief description of MPI and OpenMP and their most useful routines.

1.5.8 Appendix B - Stan

Appendix B gives brief details about Stan, its syntax and its most important functionalities.

1.5.9 Appendix C - How to Install SMC methods in Stan

Appendix C illustrates how to install a PF and an SMC sampler in Stan in order to respectively set up Streaming-Stan and SMC-Stan, the HPC extensions packages for Stan which are presented in Chapter 6.

1.5.10 Appendix D - Reversible and Symplectic Numerical Integrators: Properties

Appendix D proves an important property of Leapfrog, the numerical integrator used in NUTS and HMC. This property is fundamental to design the novel proposal distributions presented in Chapter 6.

1.6 Notation

Many papers are cited in this work and, although they mostly share the same topic, the notation tends to be very diverse and often hard to translate from one paper to another. Since the reader may be used to a different notation, it is useful to provide a section that explains the notation of this thesis and the reason behind each choice. The goal is to pick a notation which is elegant, simple and as close to the core papers of this thesis as possible, without losing sight of the main subject: presenting a novel parallel resampling algorithm for SMC methods in C/C++ programming language.

Upper-case italic here is used for scalar parameters that are often input arguments to the algorithms presented in this work. Typical examples are the number of particles N, the dimension of each particle, M, the number of MPI cores, P, or the number of SMC iterations, T_{SMC}.

Lower-case italic here is mostly used for iterators of vectors and matrices. This is a common choice in mathematics but also in computer science, including software development. Lower-case italic is also occasionally used to describe other scalar physical constants or parameters, such as the rank of an MPI core, p, or the number of particles per each core, n.

Vectors and matrices are in bold. Typical examples are $\mathbf{x}_t$, the matrix of multidimensional particles at the t-th SMC iteration, or $\mathbf{w}_t$, the vector of importance weights at the t-th SMC iteration. In order to refer to the i-th element of a vector/matrix within the same SMC iteration, the iterator i is placed in the superscript: for example $\mathbf{x}_t^i$ is the i-th particle at the time step t and $\mathbf{w}_t^i$ is its weight. Also t may be omitted for brevity if the vector is constant over time, or we are not interested in comparing the values of that vector across two or more consecutive time steps. Here it is useful to specify that elements of a vector/matrix are in bold too, although they might as well be scalars, e.g. $\mathbf{w}_t^i \in \mathbb{R}$. This choice is motivated by two reasons: first, in the case of matrices the i-th element is itself a vector (e.g. $\mathbf{x}_t^i \in \mathbb{R}^M$); second, the common view among software developers (including C/C++ developers) is to consider elements of a vector as single-element vectors themselves. It is also useful to specify that sometimes papers write iterators between brackets to avoid confusion with the power operation. Although this choice is robust and consistent, here it would be quite excessive, because lots of vector equations in this work are in element-wise form but they very rarely need powers. In those rare cases, the power of an element is expressed using brackets as follows: $(\mathbf{w}_t^i)^2$. Exceptions are made for the power of scalars or constant numbers, e.g. $2^3 = 8$.

Iterators of vectors, matrices or mathematical operations over arrays such as Sum or Product start from 0. This choice is mostly motivated by the syntax in C/C++, the main programming language here, although it is rather unpopular in mathematics and invalid in some programming languages such as MATLAB and Stan.

Chapter 2

Technical Background

This chapter provides the necessary technical background about SMC methods, with a particular emphasis on Particle Filters (PFs) and Fixed-Lag SMC, two methods that will be considered in the following chapters. Section 2.1.3 offers a brief explanation about SMC samplers, and their similarities and differences with PFs. Understanding these similarities and differences is important to understand Fixed-Lag SMC more deeply. Although this work is not about Markov Chain Monte Carlo (MCMC) methods, some novelty which is discussed in Chapter 6 requires a high level of understanding about MCMC. Therefore, Section 2.2 describes Metropolis-Hastings, Hamiltonian Monte Carlo (HMC) and No-U-Turn Sampler (NUTS), three popular MCMC methods. The reader is referred to [6, 28, 30, 31, 37, 43, 44, 61, 68] for further details.

2.1 Sequential Monte Carlo Methods

Let $\mathbf{X}_t \in \mathbb{R}^M$ be the state of a system at any given time step t. In simple terms, the state is a set of physical quantities that fully describes the most important features of the system over time. $\mathbf{X}_t$ can be evaluated by a linear or non-linear vectorial function $\mathbf{f}_t : \mathbb{R}^M \times \mathbb{R}^{M_v} \rightarrow \mathbb{R}^M$ as follows:

$$\mathbf{X}_t = \mathbf{f}_t(\mathbf{X}_{t-1}, \mathbf{V}_t) \tag{2.1}$$

where $\mathbf{V}_t \in \mathbb{R}^{M_v}$ is an i.i.d process noise. We are interested in evaluating accurately $\mathbf{X}_t$ $\forall t$. However, in many applications measurements of the state (or at least a subset of the state) are not available and samples of $\mathbf{V}_t$ cannot be generated directly. At each time step, we are only given partial measurements of $\mathbf{X}_t$, or measurements of physical quantities which are related to $\mathbf{X}_t$ by some physical law. Let $\mathbf{Y}_t \in \mathbb{R}^{M_y}$ be the measurement vector collected at t, and related to the state by a linear or non-linear vectorial function $\mathbf{g}_t : \mathbb{R}^M \times \mathbb{R}^{M_w} \rightarrow \mathbb{R}^{M_y}$ as follows:

$$\mathbf{Y}_t = \mathbf{g}_t(\mathbf{X}_t, \mathbf{W}_t) \tag{2.2}$$

where $\mathbf{W}_t \in \mathbb{R}^{M_w}$ is an i.i.d measurement noise. Equations (2.1) and (2.2) define the system model. Bayesian inference methods can be used to address this problem.

The Bayesian approach starts by defining the posterior (or target) $p(\mathbf{X}_t|\mathbf{Y}_{1:t})$, where $\mathbf{Y}_{1:t}$ is the history of measurements from time step 1 to the current time step t. The initial condition is that $p(\mathbf{X}_0|\mathbf{Y}_0) = p(\mathbf{X}_0)$ which means that no measurement is available at the beginning. Then, $p(\mathbf{X}_t|\mathbf{Y}_{1:t})$ is recursively evaluated $\forall t$ by using a prediction-update approach as follows.

In the prediction phase, the posterior at the previous time step $p(\mathbf{X}_{t-1}|\mathbf{Y}_{1:t-1})$ is assumed to be given. Then, the state-transition equation (2.1) is used to make a state prediction, which is equivalent to evaluating $p(\mathbf{X}_t|\mathbf{Y}_{1:t-1})$, the probability of $\mathbf{X}_t$ prior to collecting a new $\mathbf{Y}_t$ (commonly called prior). This can be done by using the Chapman–Kolmogorov equation as follows:

$$\begin{aligned} p(\mathbf{X}_t|\mathbf{Y}_{1:t-1}) &= \int p(\mathbf{X}_t|\mathbf{X}_{t-1}, \mathbf{Y}_{1:t-1}) p(\mathbf{X}_{t-1}|\mathbf{Y}_{1:t-1}) d\mathbf{X}_{t-1} \\ &= \int p(\mathbf{X}_t|\mathbf{X}_{t-1}) p(\mathbf{X}_{t-1}|\mathbf{Y}_{1:t-1}) d\mathbf{X}_{t-1} \end{aligned} \tag{2.3}$$

where $p(\mathbf{X}_t|\mathbf{X}_{t-1}) = p(\mathbf{X}_t|\mathbf{X}_{t-1}, \mathbf{Y}_{1:t-1})$ because (2.1) is a Markov process of order one.

Once the new measurement $\mathbf{Y}_t$ is collected, the prediction can be updated by using the Bayes' rule as follows:

$$p(\mathbf{X}_t|\mathbf{Y}_{1:t}) = \frac{p(\mathbf{X}_t|\mathbf{Y}_{1:t-1}) p(\mathbf{Y}_t|\mathbf{X}_t)}{p(\mathbf{Y}_t|\mathbf{Y}_{1:t-1})} = \frac{p(\mathbf{X}_t|\mathbf{Y}_{1:t-1}) p(\mathbf{Y}_t|\mathbf{X}_t)}{\int p(\mathbf{X}_t|\mathbf{Y}_{1:t-1}) p(\mathbf{Y}_t|\mathbf{X}_t) d\mathbf{X}_t} \tag{2.4}$$

where $p(\mathbf{Y}_t|\mathbf{X}_t)$ and $p(\mathbf{Y}_t|\mathbf{Y}_{1:t-1})$ are commonly called likelihood and evidence (or normalising constant) respectively. Equations (2.3) and (2.4) form the basis of optimal Bayesian solutions. However, this problem is often impossible to solve analytically, since the evidence may be highly dimensional or impossible to integrate.

SMC methods represent an approximate solution to (2.3) and (2.4). As mentioned in Chapter 1, PFs and SMC samplers are both SMC methods which are respectively used in the context of dynamic models and static models. Therefore, the two methods can be applied in complementary domains, although in some cases SMC Samplers can also be configured to offer improved performance in contexts where a PF struggles [57]. The next two sections and Section 2.1.4 illustrate two types of PFs, and Section 2.1.3 offers a brief description of SMC samplers.

2.1.1 Sequential Importance Sampling

SMC methods apply the Importance Sampling (IS) principle to make Bayesian inferences of the state of a dynamic or static system. The main idea consists of randomly generating $\mathbf{x}_t \in \mathbb{R}^{N \times M}$, a population of N statistically independent hypothesis of $\mathbf{X}_t$ called particles (or samples), in order to approximate the true posterior at any time step t. Each particle $\mathbf{x}_t^i$ is then assigned to an unnormalised weight $\mathbf{w}_t^i$, such that the array of weights $\mathbf{w}_t \in \mathbb{R}^N$

provides information on which particle best resembles the true state. A new estimate of $\mathbf{X}_t$ can then be computed as weighted mean of the particles. Details on IS follow.

Let $p(\mathbf{X})$ be a Probability Density Function (PDF) from which it is hard to sample directly but can still be evaluated up to a constant factor. The IS principle says that one can arbitrarily pick an alternative probability distribution $q(\mathbf{X})$, called proposal or importance density, from which it is easy to draw N samples $\mathbf{x}^i$ and give them a weight $\mathbf{w}^i$ such that:

$$p(\mathbf{X}) \approx \sum_{i=0}^{N-1} \mathbf{w}^i \delta(\mathbf{X} - \mathbf{x}^i) \propto \sum_{i=0}^{N-1} \tilde{\mathbf{w}}^i \delta(\mathbf{X} - \mathbf{x}^i) \tag{2.5}$$

where $\tilde{\mathbf{w}}^i$ is the normalised weight such that $\sum \tilde{\mathbf{w}}^i = 1$. The choice of the weights is crucial for (2.5) to work. In IS the weights are chosen up to a normalising constant factor as follows:

$$\mathbf{w}^i = \frac{p(\mathbf{x}^i)}{q(\mathbf{x}^i)} \tag{2.6}$$

Now we can apply the concepts above to the case of SMC.

Let $p(\mathbf{X}_{0:t}|\mathbf{Y}_{1:t})$ be the posterior distribution of interest where $\mathbf{X}_{0:t}$ and $\mathbf{Y}_{1:t}$ are the history of states and measurements up to time step t. According to the IS principle, we can sample N random particle trajectories $\mathbf{x}^i_{0:t}$ from the proposal $q(\mathbf{X}_{0:t}|\mathbf{Y}_{1:t})$ and compute each weights as

$$\mathbf{w}^i_t = \frac{p(\mathbf{x}^i_{0:t}|\mathbf{Y}_{1:t})}{q(\mathbf{x}^i_{0:t}|\mathbf{Y}_{1:t})} \tag{2.7}$$

such that

$$p(\mathbf{X}_{0:t}|\mathbf{Y}_{1:t}) \approx \sum_{i=0}^{N-1} \mathbf{w}^i_t \delta(\mathbf{X}_{0:t} - \mathbf{x}^i_{0:t}) \propto \sum_{i=0}^{N-1} \tilde{\mathbf{w}}^i_t \delta(\mathbf{X}_{0:t} - \mathbf{x}^i_{0:t}) \tag{2.8}$$

Now, instead of considering all time steps at every iteration, one could apply recursively the prediction-update approach of optimal Bayesian solutions. Therefore, we assume to have access to the N samples that approximate the prior $p(\mathbf{X}_{0:t-1}|\mathbf{Y}_{1:t-1})$ and we want to generate new samples out of the old ones to approximate $p(\mathbf{X}_{0:t}|\mathbf{Y}_{1:t})$. If the proposal is chosen such that it satisfies the following property

$$q(\mathbf{X}_{0:t}|\mathbf{Y}_{1:t}) = q(\mathbf{X}_t|\mathbf{X}_{0:t-1}, \mathbf{Y}_{1:t}) q(\mathbf{X}_{0:t-1}|\mathbf{Y}_{1:t-1}) \tag{2.9}$$

one could simply propagate the old particle population $\mathbf{x}_{0:t-1}$ by drawing each $\mathbf{x}^i_t$ from $q(\mathbf{X}_t|\mathbf{X}_{0:t-1}, \mathbf{Y}_{1:t})$. To derive the expression for the weights, we start from Bayes's rule and then express the posterior in terms of $p(\mathbf{Y}_t|\mathbf{X}_t)$, $p(\mathbf{X}_t|\mathbf{X}_{t-1})$ and $p(\mathbf{X}_{0:t-1}|\mathbf{Y}_{1:t-1})$.

$$\begin{aligned}
p(\mathbf{X}_{0:t}|\mathbf{Y}_{1:t}) &= \frac{p(\mathbf{Y}_t|\mathbf{X}_{0:t}, \mathbf{Y}_{1:t-1}) p(\mathbf{X}_{0:t}|\mathbf{Y}_{1:t-1})}{p(\mathbf{Y}_t|\mathbf{Y}_{1:t-1})} \\
&= \frac{p(\mathbf{Y}_t|\mathbf{X}_{0:t}, \mathbf{Y}_{1:t-1}) p(\mathbf{X}_t|\mathbf{X}_{0:t-1}, \mathbf{Y}_{1:t-1})}{p(\mathbf{Y}_t|\mathbf{Y}_{1:t-1})} p(\mathbf{X}_{0:t-1}|\mathbf{Y}_{1:t-1}) \\
&= \frac{p(\mathbf{Y}_t|\mathbf{X}_t) p(\mathbf{X}_t|\mathbf{X}_{t-1})}{p(\mathbf{Y}_t|\mathbf{Y}_{1:t-1})} p(\mathbf{X}_{0:t-1}|\mathbf{Y}_{1:t-1}) \\
&\propto p(\mathbf{Y}_t|\mathbf{X}_t) p(\mathbf{X}_t|\mathbf{X}_{t-1}) p(\mathbf{X}_{0:t-1}|\mathbf{Y}_{1:t-1})
\end{aligned} \tag{2.10}$$

If we replace the numerator and the denominator in (2.7) by using (2.9) and (2.10) and change the the state terms with particle terms, we obtain:

$$\begin{aligned}
\mathbf{w}_t^i &= \frac{p(\mathbf{Y}_t|\mathbf{x}_t^i)p(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i)p(\mathbf{x}_{0:t-1}^i|\mathbf{Y}_{1:t-1})}{q(\mathbf{x}_t^i|\mathbf{x}_{0:t-1}^i,\mathbf{Y}_{1:t})q(\mathbf{x}_{0:t-1}^i|\mathbf{Y}_{1:t-1})} \\
&= \frac{p(\mathbf{Y}_t|\mathbf{x}_t^i)p(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i)p(\mathbf{x}_{0:t-1}^i|\mathbf{Y}_{1:t-1})}{q(\mathbf{x}_t^i|\mathbf{x}_{0:t-1}^i,\mathbf{Y}_{1:t})q(\mathbf{x}_{0:t-1}^i|\mathbf{Y}_{1:t-1})}\frac{\mathbf{w}_{t-1}^i}{\mathbf{w}_{t-1}^i} \\
&= \mathbf{w}_{t-1}^i\frac{p(\mathbf{Y}_t|\mathbf{x}_t^i)p(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i)}{q(\mathbf{x}_t^i|\mathbf{x}_{0:t-1}^i,\mathbf{Y}_{1:t})}
\end{aligned} \tag{2.11}$$

Finally, if we assume that the new particles $\mathbf{x}_t^i$ are only generated based on the knowledge of $\mathbf{x}_{t-1}^i$ and $\mathbf{Y}_t$, which is the case for many SMC methods, we can then discard each $\mathbf{x}_{0:t-2}^i$ trajectory and the first $t-1$ measurements $\mathbf{Y}_{1:t-1}$. Therefore, each new particle is sampled as follows:

$$\mathbf{x}_t^i \sim q(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i,\mathbf{Y}_t) \quad \forall i = 0,1,\dots,N-1 \tag{2.12}$$

and its weight is computed as

$$\mathbf{w}_t^i = \mathbf{w}_{t-1}^i\frac{p(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i)p(\mathbf{Y}_t|\mathbf{x}_t^i)}{q(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i,\mathbf{Y}_t)} \quad \forall i = 0,1,\dots,N-1 \tag{2.13}$$

Then, the weights are normalised

$$\tilde{\mathbf{w}}_t^i = \frac{\mathbf{w}_t^i}{\sum_{j=0}^{N-1}\mathbf{w}_t^j} \quad \forall i = 0,1,\dots,N-1 \tag{2.14}$$

such that estimates of the true state can be computed at each time step as weighted mean of the particles

$$\xi_t = E(\mathbf{X}_t) = \sum_{i=0}^{N-1}\mathbf{x}_t^i\tilde{\mathbf{w}}_t^i \tag{2.15}$$

A PF which computes Equations (2.12), (2.13), (2.14) and (2.15) in sequence is normally referred to as Sequential Importance Sampling (SIS). Although the population of particles converges to the true posterior for $N \to \infty$ [6], it is also proven that SIS could only work for unfeasible values of N, as shown both theoretically and empirically in [30]. This is because the particles are subjected to a phenomenon called degeneracy which (within a few iterations) makes all weights but one decrease towards 0. The variance of the weights is indeed proven to increase at every iteration [6, 30]. Therefore, ξ_t might be very inaccurate if degeneracy is not corrected.

There exist different strategies to tackle degeneracy. This thesis only focuses on the most popular and flexible alternative which consists of performing a correction step called resampling, which removes the particles with low weight and substitutes them with copies of the particles with high weights.

2.1.2 Sequential Importance Resampling

This section provides a brief description of Sequential Importance Resampling (SIR), a widely popular extension to SIS which tackles the particle degeneracy by using resampling [6, 30, 43]. Algorithm 1 illustrate a pseudo-code for SIR which the reader is referred to during the explanation.

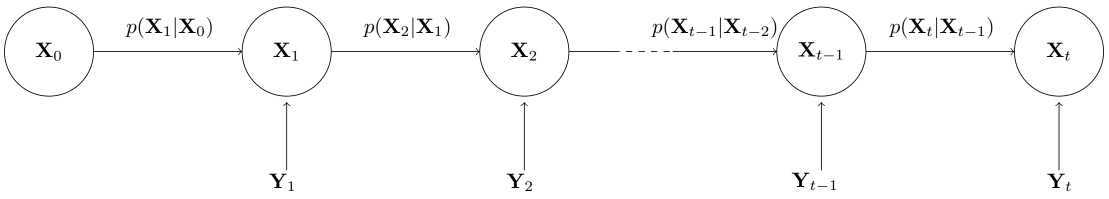


FIGURE 2.1: Sequential Importance Resampling: state flow

Algorithm 1 SIR PF

Input: T_{PF}, N, N^*

Output: ξ_t

```
1:  $\mathbf{x}_0, \mathbf{w}_0 \leftarrow$ Initialisation(), $\mathbf{x}_0 \sim p(\mathbf{x}_0)$ and $\mathbf{w}_0^i \leftarrow \frac{1}{N}\ \forall i$
2:  for $t \leftarrow 1; t \leq T_{PF}; t \leftarrow t+1$ do
3:      $\mathbf{Y}_t \leftarrow$ New_Measurement()
4:      $\mathbf{x}_t, \mathbf{w}_t \leftarrow$ IS($\mathbf{x}_{t-1}, \mathbf{w}_{t-1}, \mathbf{Y}_t$), see (2.12) and (2.13)
5:      $\tilde{\mathbf{w}}_t \leftarrow$ Normalise($\mathbf{w}_t$), see (2.14)
6:      $N_{eff} \leftarrow$ ESS($\tilde{\mathbf{w}}_t$), see (2.16)
7:      if $N_{eff} < N^*$ then Resampling
8:          ncopies $\leftarrow$ MVR($\tilde{\mathbf{w}}_t$), see (2.18)
9:          $\mathbf{x}_t \leftarrow$ Redistribute($N$, ncopies, $\mathbf{x}_t$)
10:         $\mathbf{w}_t \leftarrow$ Reset($\mathbf{w}_t$), $\mathbf{w}_t^i \leftarrow \frac{1}{N}\ \forall i$
11:     end if
12:     $\xi_t \leftarrow$ Estimate($\mathbf{x}_t$), see (2.19)
13: end for
```

In the SIR PF, at the initial time $t = 0$, no measurement has been collected yet, so the particles are initially drawn from the initial distribution $q(\mathbf{x}_0) = p(\mathbf{x}_0)$ and weighted equally to $1/N$, as this is the best assumption without feedback. However, for any time step $t > 0$ measurements are collected, and each particle is drawn from the proposal distribution as in (2.12). The importance weights are then computed as in (2.13) and normalised by using (2.14). Therefore, up to this point SIR is indistinguishable from SIS.

In order to address degeneracy, the SIR PF performs a resampling step which repopulates the particles by eliminating the most negligible ones and duplicating the most important ones. In the original Bootstrap PF, resampling is performed every iteration but in the SIR PF it is only triggered when it is needed, more precisely when the (approximate) Effective Sample Size (ESS)

$$N_{eff} = \frac{1}{\sum_{i=0}^{N-1} (\tilde{\mathbf{w}}_t^i)^2} \tag{2.16}$$

decreases below an arbitrary threshold N^*, which is commonly set to $\frac{N}{2}$.

Different (biased or unbiased) resampling schemes exist [43, 48, 49, 62] but they mostly follow a three-step approach. The first step is to process the normalised weights $\tilde{\mathbf{w}}_t$ to generate $\mathbf{ncopies} \in \mathbb{Z}^N$ such that $\mathbf{ncopies}^i$ indicates how many copies of the i-th particle must be created. Therefore, it is easy to infer that

$$\sum_{i=0}^{N-1} \mathbf{ncopies}^i = N \tag{2.17}$$

Algorithm 2 Sequential Redistribute (S-R)

```
Input: x, ncopies, N
Output: x_new
1: $i \leftarrow 0$
2: for $j \leftarrow 0; j < N; j \leftarrow j+1$ do
3:     for $k \leftarrow 0; k < \mathbf{ncopies}^j; k \leftarrow k+1$ do
4:         $\mathbf{x}_{new}^i \leftarrow \mathbf{x}^j$
5:         $i \leftarrow i+1$
6:     end for
7: end for
```

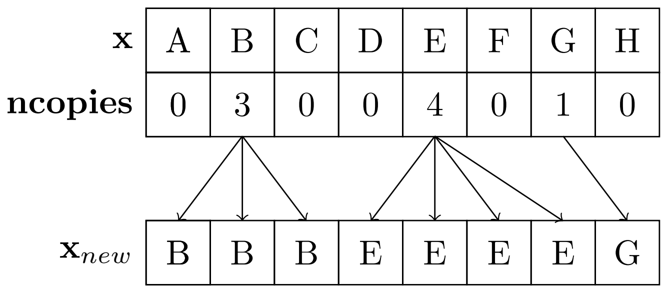


FIGURE 2.2: Sequential Redistribute

The second step is redistribution which all resampling algorithms have in common and is in charge of duplicating each particle the right number of times. A textbook Sequential Redistribute (S-R) can be found in Algorithm 2 which takes $O(N)$ steps as (2.17) holds. A possible example for $N = 8$ particles is illustrated in Figure 2.2 where the value of each particle $\mathbf{x}^i$ is actually a vector of real numbers, although it is marked with a capital letter for brevity. After redistributing, all weights are reset to $1/N$. To perform the first step, previous referenced work [6, 53, 89] has used Minimum Variance Resampling (MVR), commonly known as Systematic Resampling [43]. Since this thesis focuses mostly on redistribution and its implementation on a parallel architecture, MVR will be the only variant considered. MVR first computes $\mathbf{cdf} \in \mathbb{R}^{N+1}$, the Cumulative

Density Function (CDF) of the weights, then it draws a random sample $u \sim [0,1)$ from a uniform distribution and then computes each $\mathbf{ncopies}^i$ as follows:

$$\mathbf{ncopies}^i = \lceil \mathbf{cdf}^{i+1} - u \rceil - \lceil \mathbf{cdf}^i - u \rceil \quad \forall i = 0, 1, \dots, N-1 \tag{2.18}$$

where the bracket operator represents the ceiling function (e.g. $\lceil 5.1 \rceil = 6$). At the end of each time step, estimates are produced as follows:

$$\xi_t = E(\mathbf{X}_t) = \frac{1}{N} \sum_{i=0}^{N-1} \mathbf{x}_t^i \tag{2.19}$$

2.1.3 Sequential Monte Carlo Samplers

In the SMC samplers, we have no income stream of measurements $\mathbf{Y}_t$ during the SMC iterations, but all data $\mathbf{Y}$ is assumed to be given before each run [61]. Therefore, at each iteration t we consider a static target distribution $p(\mathbf{X}_t|\mathbf{Y})$. The objective is to draw samples from $p(\mathbf{X}_t|\mathbf{Y})$ to estimate the true state $\mathbf{X}_t$ at the final SMC iteration $t = T_{SMC}$. This is because the target is static and hence every new estimate is simply an improved version of the previous one. In order to do that, one begins by defining the joint distribution over the state trajectory $\mathbf{X}_{0:t}$ as follows:

$$p(\mathbf{X}_{0:t}|\mathbf{Y}) = p(\mathbf{X}_t|\mathbf{Y}) \prod_{\tau=1}^{t} L(\mathbf{X}_{\tau-1}|\mathbf{X}_\tau) \tag{2.20}$$

where the backward Markov kernel $L(\mathbf{X}_{\tau-1}|\mathbf{X}_\tau)$ is arbitrary and defined such that

$$\int p(\mathbf{X}_{0:t}|\mathbf{Y}) d\mathbf{X}_{0:t-1} = p(\mathbf{X}_t|\mathbf{Y}) \tag{2.21}$$

Now we apply the IS principle. Therefore, at each iteration we draw a population of N samples $\mathbf{x}_t^i$ from a proposal distribution $q(\mathbf{X}_{0:t}) = q(\mathbf{X}_t|\mathbf{X}_{t-1})q(\mathbf{X}_{0:t-1})$ and give them an importance weight $\mathbf{w}_t^i = \frac{p(\mathbf{x}_{0:t}^i|\mathbf{Y})}{q(\mathbf{x}_{0:t}^i)}$. This means that the samples are first generated from the initial proposal $q_0(\mathbf{X}_0)$ and given the initial weight $\mathbf{w}_0^i = \frac{p(\mathbf{x}_0^i|\mathbf{Y})}{q_0(\mathbf{x}_0^i)}$. At any iteration $t > 0$, each particle is drawn from the proposal distribution as follows:

$$\mathbf{x}_t^i \sim q(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i) \quad \forall i = 0, 1, \dots, N-1 \tag{2.22}$$

and its weight is computed as

$$\begin{aligned} \mathbf{w}_t^i = \frac{p(\mathbf{x}_{0:t}^i|\mathbf{Y})}{q(\mathbf{x}_{0:t}^i)} &= \frac{p(\mathbf{x}_t^i|\mathbf{Y}) \prod_{\tau=1}^{t} L(\mathbf{x}_{\tau-1}^i|\mathbf{x}_\tau^i)}{q(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i)q(\mathbf{x}_{0:t}^i)} \\ &= \frac{p(\mathbf{x}_t^i|\mathbf{Y}) \prod_{\tau=1}^{t} L(\mathbf{x}_{\tau-1}^i|\mathbf{x}_\tau^i)}{q(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i)q(\mathbf{x}_{0:t}^i)} \frac{\mathbf{w}_{t-1}^i}{\mathbf{w}_{t-1}^i} \\ &= \mathbf{w}_{t-1}^i \frac{p(\mathbf{x}_t^i|\mathbf{Y})}{p(\mathbf{x}_{t-1}^i|\mathbf{Y})} \frac{L(\mathbf{x}_{t-1}^i|\mathbf{x}_t^i)}{q(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i)} \quad \forall i = 0, 1, \dots, N-1 \end{aligned} \tag{2.23}$$

As we can see, SMC samplers work on static models as (2.23) does not depend on an on-line stream of data which, on the other hand, impacts the weights for PFs.

After IS, the samples are again normalised as in (2.14) and estimates can then be evaluated as in (2.15). After that, a resampling step may be performed depending on the value of N_{eff}. In the vanilla SMC Sampler, these steps are performed iteratively until we reach the final iteration $t = T_{SMC}$.

In [65], a novel recycling method is proposed. Instead of considering the samples from the last iteration as providing the outputs, estimates are computed using all particles from all iterations. Using the notation of this thesis, the final estimates are performed as follows:

$$\hat{\xi}_t = \frac{\sum_{\tau=1}^{t} \xi_\tau \tilde{\mathbf{c}}_\tau}{\sum_{\tau=1}^{t} \tilde{\mathbf{c}}_\tau} \quad \forall t = 1, 2, \ldots, T_{SMC} \tag{2.24}$$

where the normalisation constants $\tilde{\mathbf{c}}_t$ are computed during the SMC iterations as:

$$\tilde{\mathbf{c}}_t = \int p(\mathbf{X}_t) d\mathbf{X}_{0:t-1} \approx \mathbf{c}_t = \frac{\sum_{i=0}^{N-1} \mathbf{w}_t^i}{\sum_{i=0}^{N-1} \mathbf{w}_{t-1}^i} \tag{2.25}$$

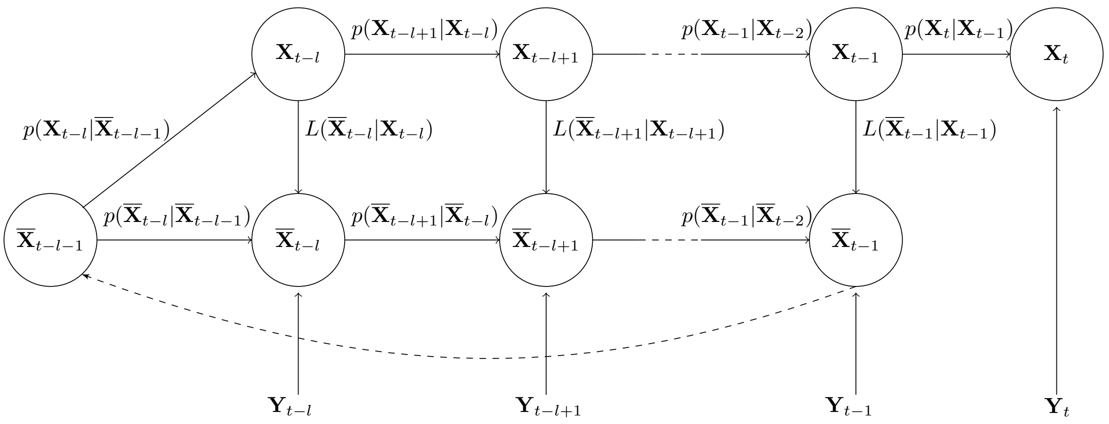
Algorithm 3 describes the SMC Sampler with the recycling method.

Algorithm 3 SMC sampler with recycling

```
Input:  $T_{SMC}$, $N$, $N^*$
Output:  $\hat{\xi}$
 1: $\mathbf{x}_0, \mathbf{w}_0 \leftarrow$ Initialisation(), $\mathbf{x}_0^i \sim q(\mathbf{x}_0^i)$, $\mathbf{w}_0^i = \frac{p_0(\mathbf{x}_0^i|\mathbf{Y})}{q_0(\mathbf{x}_0^i)}$ $\forall i$
 2: for $t \leftarrow 1$; $t \leq T_{SMC}$; $t \leftarrow t+1$ do
 3:     $\tilde{c}_t \leftarrow$ Normalisation_Constant($\mathbf{w}_t$), see (2.25)
 4:     $\mathbf{x}_t, \mathbf{w}_t \leftarrow$ IS($\mathbf{x}_{t-1}, \mathbf{w}_{t-1}$), see (2.22) and (2.23)
 5:     $\tilde{\mathbf{w}}_t \leftarrow$ Normalise($\mathbf{w}_t$), see (2.14)
 6:     $\xi_t \leftarrow$ Weighted_Mean($\mathbf{x}_t, \mathbf{w}_t$), see (2.15)
 7:     $N_{eff} \leftarrow$ ESS($\tilde{\mathbf{w}}_t$), see (2.16)
 8:     if $N_{eff} < N^*$ then Resampling
 9:         ncopies $\leftarrow$ MVR($\tilde{\mathbf{w}}_t$), see (2.18)
10:         $\mathbf{x}_t \leftarrow$ Redistribute($N$, ncopies, $\mathbf{x}_t$)
11:         $\mathbf{w}_t \leftarrow$ Reset($\mathbf{w}_t$), $\mathbf{w}_t^i \leftarrow \frac{1}{N}$ $\forall i$
12:     end if
13: end for
14: $\hat{\xi} \leftarrow$Recycling($\xi, \tilde{\mathbf{c}}, T_{SMC}$), see (2.24)
```

2.1.4 Fixed-Lag Sequential Monte Carlo

Fixed-Lag SMC is a special type of PF that was first presented in [28]. Once again IS is applied to approximate the PDF of $\mathbf{X}_t$, but in this case $l+1 \in \mathbb{Z}^+$ measurements $\mathbf{Y}_{t-l:t}$ are considered per time step, where $l \in \mathbb{Z}$ is a fixed lag. This means one can re-sample the values of each particle from step $t-l$ to $t-1$, such that each new particle $\mathbf{x}_t^i$ can be generated given a corrected (and potentially improved) particle trajectory $\mathbf{x}_{t-l:t-1}^i$.



$\mathbf{X}_{t-l}$
$\mathbf{X}_{t-l+1}$
$\mathbf{X}_{t-1}$
$\mathbf{X}_{t}$
$p(\mathbf{X}_{t-l+1}|\mathbf{X}_{t-l})$
$p(\mathbf{X}_{t-1}|\mathbf{X}_{t-2})$
$p(\mathbf{X}_{t}|\mathbf{X}_{t-1})$
$p(\mathbf{X}_{t-l}|\overline{\mathbf{X}}_{t-l-1})$
$L(\overline{\mathbf{X}}_{t-l}|\mathbf{X}_{t-l})$
$L(\overline{\mathbf{X}}_{t-l+1}|\mathbf{X}_{t-l+1})$
$L(\overline{\mathbf{X}}_{t-1}|\mathbf{X}_{t-1})$
$\overline{\mathbf{X}}_{t-l-1}$
$\overline{\mathbf{X}}_{t-l}$
$\overline{\mathbf{X}}_{t-l+1}$
$\overline{\mathbf{X}}_{t-1}$
$p(\overline{\mathbf{X}}_{t-l}|\overline{\mathbf{X}}_{t-l-1})$
$p(\overline{\mathbf{X}}_{t-l+1}|\overline{\mathbf{X}}_{t-l})$
$p(\overline{\mathbf{X}}_{t-1}|\overline{\mathbf{X}}_{t-2})$
$\mathbf{Y}_{t-l}$
$\mathbf{Y}_{t-l+1}$
$\mathbf{Y}_{t-1}$
$\mathbf{Y}_{t}$

FIGURE 2.3: Fixed-Lag Sequential Monte Carlo: state flow

This method starts by drawing the first set of particles from the initial distribution and then drawing other $l-1$ sets as in (2.12). The same weight $1/N$ is assigned to each current particle trajectory $\overline{\mathbf{x}}^i_{0:l-1}$. At the time step $t \geq l$, we are then given a set of N particles $\overline{\mathbf{x}}^i_{0:t-1}$ and weights $\mathbf{w}^i_{t-1}$ which approximate the true posterior $p(\overline{\mathbf{X}}_{0:t-1}|\mathbf{Y}_{1:t-1})$. A new measurement $\mathbf{Y}_t$ is collected. Since the goal is to propose new particles $\mathbf{x}^i_t$ based on corrected values for $\overline{\mathbf{x}}^i_{t-l-1:t-1}$, we can then extend (2.12) for the Fixed-Lag SMC case as follows:

$$\mathbf{x}^i_{t-l:t} \sim q(\mathbf{x}^i_{t-l:t}|\overline{\mathbf{x}}^i_{t-l-1}, \mathbf{Y}_{t-l:t}) \quad \forall i = 0, 1, \dots, N-1 \tag{2.26}$$

To infer the importance weight formula, we define the posterior as follows:

$$\begin{aligned} &p(\mathbf{X}_{t-l:t}, \overline{\mathbf{X}}_{0:t-l-1}|\mathbf{Y}_{1:t}) = \\ &\qquad p(\mathbf{X}_{t-l:t}|\overline{\mathbf{X}}_{t-l-1}, \mathbf{Y}_{t-l:t})L(\overline{\mathbf{X}}_{1:r-l-1}|\overline{\mathbf{X}}_{r-l:t-1})L(\overline{\mathbf{X}}_{r-l:t-1}|\mathbf{X}_{t-l:t-1}) \end{aligned} \tag{2.27}$$

where $L(\overline{\mathbf{X}}_{1:r-l-1}|\overline{\mathbf{X}}_{r-l:t-1})$ generically summarises the product of the backward kernels that were necessary to perform the previous $t-1$ steps. On the other hand, $L(\overline{\mathbf{X}}_{r-l:t-1}|\mathbf{X}_{t-l:t-1}) = \prod_{\tau=t-l}^{t-1} L(\overline{\mathbf{X}}_\tau|\mathbf{X}_\tau)$ which is necessary to update each particle trajectory $\overline{\mathbf{x}}^i_{t-l:t-1}$ to $\mathbf{x}^i_{t-l:t-1}$ at the current time step t. The weight formula can now be inferred from (2.26) and (2.27) and by splitting the posterior terms into prior-likelihood products as follows

$$\begin{aligned} \mathbf{w}^i_t &= \frac{p(\mathbf{x}^i_{t-l:t}, \overline{\mathbf{x}}^i_{0:t-l-1}|\mathbf{Y}_{1:t})}{q(\mathbf{x}^i_{t-l:t}|\overline{\mathbf{x}}^i_{t-l-1}, \mathbf{Y}_{t-l:t})} \\ &= \frac{p(\mathbf{x}^i_{t-l:t}|\overline{\mathbf{x}}^i_{t-l-1}, \mathbf{Y}_{t-l:t})L(\overline{\mathbf{x}}_{1:r-l-1}|\overline{\mathbf{x}}_{r-l:t-1})L(\overline{\mathbf{x}}^i_{r-l:t-1}|\mathbf{x}^i_{t-l:t-1})}{q(\mathbf{x}^i_{t-l:t}|\overline{\mathbf{x}}^i_{t-l-1}, \mathbf{Y}_{t-l:t})} \\ &= \frac{p(\mathbf{x}^i_{t-l:t}|\overline{\mathbf{x}}^i_{t-l-1}, \mathbf{Y}_{t-l:t})L(\overline{\mathbf{x}}_{1:r-l-1}|\overline{\mathbf{x}}_{r-l:t-1})L(\overline{\mathbf{x}}^i_{r-l:t-1}|\mathbf{x}^i_{t-l:t-1})}{q(\mathbf{x}^i_{t-l:t}|\overline{\mathbf{x}}^i_{t-l-1}, \mathbf{Y}_{t-l:t})} \frac{\mathbf{w}^i_{t-1}}{\mathbf{w}^i_{t-1}} \\ &= \mathbf{w}^i_{t-1} \frac{p(\mathbf{x}^i_{t-l:t}|\overline{\mathbf{x}}^i_{t-l-1}, \mathbf{Y}_{t-l:t})L(\overline{\mathbf{x}}^i_{r-l:t-1}|\mathbf{x}^i_{t-l:t-1})}{p(\overline{\mathbf{x}}^i_{t-l:t-1}|\overline{\mathbf{x}}^i_{t-l-1}, \mathbf{Y}_{t-l:t-1})q(\mathbf{x}^i_{t-l:t}|\overline{\mathbf{x}}^i_{t-l-1}, \mathbf{Y}_{t-l:t})} \end{aligned}$$

$$= \mathbf{w}_{t-1}^i \frac{p(\mathbf{x}_{t-l:t}^i|\overline{\mathbf{x}}_{t-l-1}^i)p(\mathbf{Y}_{t-l:t}|\mathbf{x}_{t-l:t}^i)L(\overline{\mathbf{x}}_{t-l:t-1}^i|\mathbf{x}_{t-l:t-1}^i)}{p(\overline{\mathbf{x}}_{t-l:t-1}^i|\overline{\mathbf{x}}_{t-l-1}^i)p(\mathbf{Y}_{t-l:t-1}|\overline{\mathbf{x}}_{t-l:t}^i)q(\mathbf{x}_{t-l:t}^i|\overline{\mathbf{x}}_{t-1}^i, \mathbf{Y}_{t-l:t})} \quad \forall i = 0, 1, \ldots, N-1 \tag{2.28}$$

After that, the Fixed-Lag SMC method performs the same operations as SIR PF, in other words, weight normalisation as in (2.14), resampling according to (2.16) and then produces state estimates as in (2.19). Algorithm 4 depicts a pseudo-code for the Fixed-Lag SMC method.

Algorithm 4 Fixed-Lag SMC

Input: T_{PF}, N, N^*, l

Output: ξ_t

```
1: $\mathbf{x}_0, \mathbf{w}_0 \leftarrow$ Initialisation(), $\mathbf{x}_0 \sim p(\mathbf{x}_0)$ and $\mathbf{w}_0^i \leftarrow \frac{1}{N}\ \forall i$
2: for $t \leftarrow 1; t \leq l-1; t \leftarrow t+1$ do
3:     $\mathbf{Y}_t \leftarrow$ New_Measurement()
4:     $\mathbf{x}_t^i \sim p(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i, \mathbf{Y}_t)\ \forall i$
5:     $\mathbf{w}_t^i \leftarrow \frac{1}{N}\ \forall i$
6: end for
7: for $t \leftarrow l; t \leq T_{PF}; t \leftarrow t+1$ do
8:     $\mathbf{Y}_t \leftarrow$ New_Measurement()
9:     $\mathbf{x}_{t-l:t}, \mathbf{w}_t \leftarrow$ IS($\mathbf{x}_{t-l-1:t-1}, \mathbf{w}_{t-1}, \mathbf{Y}_t$), see (2.26) and (2.28)
10:    $\tilde{\mathbf{w}}_t \leftarrow$ Normalise($\mathbf{w}_t$), see (2.14)
11:    $N_{eff} \leftarrow$ ESS($\tilde{\mathbf{w}}_t$), see (2.16)
12:    if $N_{eff} < N^*$ then Resampling
13:        ncopies $\leftarrow$ MVR($\tilde{\mathbf{w}}_t$), see (2.18)
14:        $\mathbf{x}_{t-l:t} \leftarrow$ Redistribute($N$, ncopies, $\mathbf{x}_{t-l:t}$)
15:        $\mathbf{w}_t \leftarrow$ Reset($\mathbf{w}_t$), $\mathbf{w}_t^i \leftarrow \frac{1}{N}\ \forall i$
16:    end if
17:    $\xi_t \leftarrow$ Estimate($\mathbf{x}_t$), see (2.19)
18: end for
```

2.2 Markov Chain Monte Carlo Methods

MCMC is a widely popular class of methodologies which can be used in the same context of SMC samplers. Therefore, the objective is again to estimate a parameter $\mathbf{X} \in \mathbb{R}^M$ after we collect some data $\mathbf{Y} \in \mathbb{R}^{M_y}$ by drawing random samples from a static posterior distribution

$$p(\mathbf{X}|\mathbf{Y}) = \frac{p(\mathbf{X})p(\mathbf{Y}|\mathbf{X})}{p(\mathbf{Y})} = \frac{p(\mathbf{X})p(\mathbf{Y}|\mathbf{X})}{\int p(\mathbf{X})p(\mathbf{Y}|\mathbf{X})dX} \tag{2.29}$$

However, while SMC samplers are population-based methods which produce all samples independently and update them during the SMC iterations, MCMC draws each sample individually based on the knowledge of the previous one. This way a Markov chain is built sequentially in order to explore all regions of the posterior.

The next three sections briefly describes three of the most popular MCMC methods and are fundamental to understand the concepts explained in Chapter 6.

2.2.1 Metropolis-Hastings

Metropolis-Hasting is an old but still widely used MCMC method. Since we know it is often impossible to compute analytically the evidence in (2.29), the key idea is to randomly draw a new sample $\mathbf{x}^* \sim q(\mathbf{X}_t|\mathbf{X}_{t-1})$ every iteration, where $q(\mathbf{X}_t|\mathbf{X}_{t-1})$ is a proposal distribution from which we can sample directly. A typical choice is to draw $\mathbf{x}^*$ from a normal distribution with mean equal to the old sample $\mathbf{x}_{t-1}$. This is why Metropolis-Hasting is a member of the sub-class of Random-Walk (RW) MCMC methods. Gibbs Sampling [33] is another MCMC method similar, in its basic implementation, to Metropolis-Hastings.

An acceptance-rejection mechanism called Rejection Sampling is applied, in order to compensate for the error introduced by sampling from the proposal. The new sample is accepted or rejected by comparing the value of (2.29) in $\mathbf{x}^*$ with the same computed in $\mathbf{x}_{t-1}$, whose ratio cancels out the evidence. More precisely, an acceptance probability $a = \min\{1, \frac{p(\mathbf{x}^*|\mathbf{Y})q(\mathbf{x}_{t-1}|\mathbf{x}^*)}{p(\mathbf{x}_{t-1}|\mathbf{Y})q(\mathbf{x}^*|\mathbf{x}_{t-1})}\}$ is computed. $\mathbf{x}^*$ is accepted or rejected depending on whether a is lower or higher than a random number drawn from a $\texttt{Uniform}[0,1]$ distribution. In the end, the first (user-defined) τ samples are discharged (burn-in) to reduce the dependency on the initial sample. These steps are summarised by Algorithm 5.

Algorithm 5 Metropolis-Hastings

```
Input: $T_{MH}$, $\epsilon$, $\mathbf{\Sigma}$, $\tau$
Output: $\mathbf{x}$
1: $\mathbf{x}_0 \sim q_0(\mathbf{x}_0)$, Draw the initial sample from the initial proposal
2: for $t \leftarrow 1$; $t \leq T_{MH}$; $t \leftarrow t+1$ do
3:     $\mathbf{x}^* \sim q(\mathbf{x}^*|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}^*|\mathbf{x}_{t-1}, \epsilon^2\mathbf{\Sigma})$, draw a new sample from the proposal
4:     $a = \min\{1, \frac{p(\mathbf{x}^*|\mathbf{Y})q(\mathbf{x}_{t-1}|\mathbf{x}^*)}{p(\mathbf{x}_{t-1}|\mathbf{Y})q(\mathbf{x}^*|\mathbf{x}_{t-1})}\}$, calculate the acceptance probability
5:     if $a < \texttt{Uniform}[0,1]$ then
6:         $\mathbf{x}_t = \mathbf{x}^*$, the proposed sample is accepted
7:     else
8:         $\mathbf{x}_t = \mathbf{x}_{t-1}$, the proposed sample is rejected
9:     end if
10: end for
11: Remove $\mathbf{x}_{0:\tau-1}$, burn-in the first $\tau$ samples
```

Pure RW methods commonly suffer from the so-called *curse of dimensionality* which causes accuracy loss as M increases. This problem can be addressed by using alternative MCMC methods such as HMC which is explained in the next section.

2.2.2 Hamiltonian Monte Carlo

Hamiltonian Monte Carlo (HMC) is a gradient-based MCMC methods, named as such because new samples are proposed by a numerical integrator computing the gradient of the target distribution, instead of taking purely random steps. This algorithm was first presented in [31] but recently improved several times [17, 18] and applied in multiple fields such as Control Theory [3], Image Processing [17] and Tracking [72, 93]. The key idea consists of applying Hamiltonian dynamics to generate a Markov chain of samples from a target distribution. More precisely, HMC uses the Hamiltonian function $H(\mathbf{X}, \mathbf{V}) = U(\mathbf{X}) + K(\mathbf{V})$ as proposal, where $U(\mathbf{X})$ represents the potential energy, $K(\mathbf{V})$ is the kinetic energy and $\mathbf{X}$ and $\mathbf{V}$ are the position and momentum vectors respectively.

The physical explanation of Hamiltonian dynamics can be found in several textbooks such as [54, 55]. In the classic example we consider a pendulum of length l to which a mass m is attached. At the beginning, the pendulum is still on the equilibrium point. However, when we poke it, the pendulum starts oscillating forwards and backwards with velocity v, forming an angle θ (which defines its position) with the axis perpendicular to the ground. The energy is equal to $H = \frac{1}{2}ml^2\dot{\theta}^2 + mlg(1 - \cos(\theta))$, where $v = \dot{\theta} = \frac{d\theta}{dt}$ and g is the gravitational constant. As the pendulum goes up, its potential energy $mlg(1 - \cos(\theta))$ increases while its kinetic energy $\frac{1}{2}ml^2\dot{\theta}^2$ decreases, until it reaches the point of maximum height $\theta = 90°$, where it stops and has maximum potential energy. Then, the pendulum starts moving backwards and keeps accelerating until it gets to the point of minimum height $\theta = 0°$, where it reaches the maximum velocity and kinetic energy but has no potential energy. Therefore, H is conserved according to the energy-conservation law. Broadly speaking, the Hamiltonian function is constant and the potential and kinetic energy are defined as follows:

$$U(\mathbf{X}) = -\log(p(\mathbf{X}|\mathbf{Y})) \tag{2.30a}$$

$$K(\mathbf{V}) = \frac{1}{2}\mathbf{V}^{\mathsf{T}}\mathbf{M}^{-1}\mathbf{V} \tag{2.30b}$$

where $\mathbf{M}$ is the mass matrix and $p(\mathbf{X}|\mathbf{Y})$ is the target distribution for the purposes of sampling.

HMC draws samples from $p(\mathbf{X}|\mathbf{Y})$ by emulating the Hamiltonian dynamics in (2.30). For each new sample, a momentum vector $\mathbf{V}$ is drawn from $\mathcal{N}(\mathbf{0}, \mathbf{M})$. Then, the following system of Partial Differential Equations (PDEs) is solved starting from the current sample $\mathbf{x}_{t-1}$:

$$\frac{d\mathbf{V}}{dt} = -\frac{\partial U(\mathbf{X})}{\partial \mathbf{X}} \tag{2.31a}$$

$$\frac{d\mathbf{X}}{dt} = \mathbf{M}^{-1}\mathbf{V} \tag{2.31b}$$

The final position after solving (2.31) is the proposed sample which is either accepted or rejected according to a Rejection Sampling decision mechanism as in Metropolis-Hasting. The acceptance probability is the ratio between the potential energy in the current position and the one evaluated in the proposed sample. This routine is repeated for T_{HMC} iterations. Then, a subset of the initial samples is burned-in as in Metropolis-Hastings.

A numerical integrator is required to solve (2.31). Leapfrog is the typical choice in this case, because it is a symplectic integrator and because of its time-reversibility, a property that will also become useful for the novel content in Chapter 6. To draw a new sample, HMC performs an arbitrary number of Leapfrog steps L, each taking a constant step-size Δh, by using the following equations:

$$\mathbf{A}_k = -\frac{\partial U(\mathbf{X}_k)}{\partial \mathbf{X}} \tag{2.32a}$$

$$\mathbf{V}_{k+\frac{1}{2}} = \mathbf{V}_{k-\frac{1}{2}} + \Delta h \mathbf{A}_k \tag{2.32b}$$

$$\mathbf{X}_{k+1} = \mathbf{X}_k + \Delta h \mathbf{V}_{k+\frac{1}{2}} \tag{2.32c}$$

where $\mathbf{A}_k$ is the acceleration term. In HMC, $\mathbf{X}_0$ is initialised to the current sample and $\mathbf{V}_0 \sim \mathcal{N}(\mathbf{0}, \mathbf{M})$. By substituting (2.30a) into (2.32a) we obtain:

$$\mathbf{A}_k = \frac{\partial \log(p(\mathbf{X}_k|\mathbf{Y}))}{\partial \mathbf{X}} \tag{2.33}$$

which means that the new sample will be updated according to the direction given by the gradient of the target distribution, computed in the current position. Algorithms 6 and 7 show pseudo-codes for the Leapfrog integrator and HMC respectively. Despite

Algorithm 6 Leapfrog

```
Input: $\mathbf{X}'$, $\mathbf{V}'$, $L$, $\Delta h$, $\mathbf{M}$, $p(\mathbf{X}|\mathbf{Y})$, $d$ ($d$ is $+1$ or $-1$ depending on the direction)
Output: $\mathbf{X}_L$, $\mathbf{V}_L$
1: $\mathbf{X}_0 \leftarrow \mathbf{X}'$
2: $\mathbf{V}_0 \leftarrow \mathbf{V}'$
3: for $k \leftarrow 0$; $k < L$; $k \leftarrow k+1$ do
4:     $\mathbf{A}_k \leftarrow \frac{\partial \log(p(\mathbf{X}_k|\mathbf{Y}))}{\partial \mathbf{X}}$
5:     $\mathbf{V}_{k+\frac{1}{2}} \leftarrow \mathbf{V}_{k-\frac{1}{2}} + \Delta h \mathbf{A}_k$
6:     $\mathbf{X}_{k+1} \leftarrow \mathbf{X}_k + \Delta h \mathbf{V}_{k+\frac{1}{2}} d$,
7: end for
```

its efficiency, HMC requires manual tuning of Δh and L. The next section describes another gradient-based MCMC method which overcomes this limitation for L.

2.2.3 No-U-Turn Sampler

NUTS is an extension to HMC which performs an adaptive number of Leapfrog steps L by removing the need to pre-calculate its optimal value. This algorithm was first presented in [44] and now is used in a wide range of application domains due to the growing popularity of Stan [34, 67, 94], PyMC3 [75] and Pyro [11].

One of the limitations in regular HMC is that a constant L may not always be the best choice. For example, in some cases the posterior could be explored further by the same Leapfrog integration but we might have chosen a low L. In some other cases, L is too big and the Markov chain starts moving backwards. The key idea in NUTS is to stop the Hamiltonian simulation when a breaking condition, called U-Turn, is met.

Algorithm 7 Hamiltonian Monte Carlo (HMC)

Input: T_{HMC}, L, Δh, $p(\mathbf{X}|\mathbf{Y})$, $\mathbf{M}$, τ

Output: $\mathbf{x}$

```
 1: $\mathbf{x}_0 \sim q_0(\mathbf{x}_0)$, Draw the initial sample from the initial proposal
 2: for $t \leftarrow 1; t \leq T_{HMC}; t \leftarrow t+1$ do
 3:     $\mathbf{v} \sim \mathcal{N}(\mathbf{0}, \mathbf{M})$, draw a random momentum
 4:     $U \leftarrow \frac{\mathbf{v}^\intercal \mathbf{M}^{-1} \mathbf{v}}{2} + p(\mathbf{x}_{t-1}|\mathbf{Y})$
 5:     $\mathbf{x}^*, \mathbf{v}^* \leftarrow \texttt{Leapfrog}(\mathbf{x}_{t-1}, \mathbf{v}, L, \Delta h, \mathbf{M}, p(\mathbf{X}|\mathbf{Y}))$, see Algorithm 6
 6:     $U^* \leftarrow \frac{\mathbf{v}^{*\intercal} \mathbf{M}^{-1} \mathbf{v}^*}{2} + p(\mathbf{x}^*|\mathbf{Y})$
 7:     $a = \exp(U^* - U)$, calculate the acceptance probability
 8:     $u \sim [0, 1]$
 9:     if $a < u$ then
10:         $\mathbf{x}_t = \mathbf{x}^*$, the proposed sample is accepted
11:     else
12:         $\mathbf{x}_t = \mathbf{x}_{t-1}$, the proposed sample is rejected
13:     end if
14: end for
15: Remove $\mathbf{x}_{0:\tau-1}$, burn-in the first $\tau$ samples
```

More precisely, NUTS stops the simulation when the (squared) distance between the current and proposed position begins to decrease, which translates to checking whether:

$$\frac{\partial}{\partial t} \frac{(\mathbf{x}_{new} - \mathbf{x}_{old})^\intercal (\mathbf{x}_{new} - \mathbf{x}_{old})}{2} = (\mathbf{x}_{new} - \mathbf{x}_{old}) \cdot \mathbf{v} < 0 \tag{2.34}$$

However, the previous condition does not guarantee time-reversibility. To overcome this limitation, NUTS incorporates Slice Sampling into a state-doubling recursive procedure. To understand what that means, this thesis now provides a brief explanation of Slice Sampling for brevity but further details can be found in [64]. According to Slice Sampling, we can augment our HMC posterior $p(\mathbf{X}, \mathbf{V}|\mathbf{Y}) = p(\mathbf{X}, \mathbf{V}) \propto p(\mathbf{X}, \mathbf{V}) = \exp(U(\mathbf{X}) - \frac{\mathbf{V}^\intercal \cdot \mathbf{V}}{2})$ by using an auxiliary slice variable $u \sim \texttt{Uniform}[0, p(\mathbf{X}, \mathbf{V})]$ such that we get:

$$p(\mathbf{X}, \mathbf{V}, u) = \mathbb{I}(u \in [0, p(\mathbf{X}, \mathbf{V})]) \tag{2.35}$$

where the function $\mathbb{I}(.)$ equals 1 or 0 depending on whether its input argument is true or false. Hence, $p(\mathbf{X}, \mathbf{V}, u)$ has the following property:

$$\int p(\mathbf{X}, \mathbf{V}, u) du \propto p(\mathbf{X}, \mathbf{V}) \tag{2.36}$$

which means we can obtain samples of the posterior from $p(\mathbf{X}, \mathbf{V}, u)$ as long as they fall within the bounds of u, and then we can forget about u.

NUTS starts the new iteration from the current sample $\mathbf{x}_{t-1}$ and generates a random momentum $\mathbf{v} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and a slice variable $u \sim \texttt{Uniform}(0, \exp(\log p(\mathbf{x}_{t-1}|\mathbf{Y}) - \frac{\mathbf{v}^\intercal \mathbf{M}^{-1} \mathbf{v}}{2}))$. Then it augments the Markov chain by taking one step forward or backward using the Hamiltonian dynamics, where the direction is picked randomly. This procedure is repeated recursively, each time by doubling the number of steps forward or backward, whose direction is always selected randomly. Therefore, NUTS recursively builds a binary tree, which keeps track of the Leapfrog steps taken. The algorithm stops when the following U-Turn condition is met when:

$$(\mathbf{x}_+ - \mathbf{x}_-) \cdot \mathbf{v}_- < 0 \quad \vee \quad (\mathbf{x}_+ - \mathbf{x}_-) \cdot \mathbf{v}_+ < 0 \tag{2.37}$$

where $\mathbf{x}_+$, $\mathbf{v}_+$ are the proposed position and velocity in the forward direction after a Leapfrog step, and $\mathbf{x}_-$, $\mathbf{v}_-$ represent the same in the backward direction. Once (2.37) occurs, a point from the binary tree is uniformly selected to be the proposed sample.

This implementation of NUTS is however inefficient because the number of target and (most importantly) gradient evaluations grow exponentially with the height of the binary tree. The same can also be said about the memory usage. This second issue has been solved in [44] by using the following sophisticated transition kernel:

$$T(\mathbf{s}^{'}|\mathbf{s}, C) = \begin{cases} \frac{\mathbb{I}(\mathbf{s}^{'} \in C_{new})}{|C_{new}|} & \text{if } |C_{new}| > |C_{old}| \\ \frac{|C_{new}|\mathbb{I}(\mathbf{s}^{'} \in C_{new})}{|C_{old}||C_{new}|} + \left(1 - \frac{|C_{new}|}{|C_{old}|}\right)\mathbb{I}(\mathbf{s}^{'} = \mathbf{s}) & \text{if } |C_{new}| \leq |C_{old}| \end{cases} \tag{2.38}$$

where C_{old} and C_{new} are the current and proposed sets of points, $\mathbf{s}$ is a generic position-velocity pair. Hence, NUTS uses (2.38) to make a transition from C_{old} to C_{new}, and then accepts or rejects depending on the acceptance probability $\frac{|C_{new}|}{|C_{old}|}$. In [44], it has been proven that NUTS can cover more space on average by using (2.38) rather than a simple uniform sampler, while also having a space complexity which grows linearly with the binary tree.

Algorithm 8 summarises the described routine. In this implementation Δh is user-defined, while [44] also illustrates an alternative implementation where the given Δh is optimised at run-time during the burn-in. The explanation is omitted for brevity, but this optimisation is applied in the experiments of Chapter 6 since the source code used for NUTS is the one available in Stan's back end.

2.2.3.1 Example: Neal's Funnel

In this example, Metropolis-Hasting, HMC and NUTS are employed to draw 100 samples from the following M-dimensional funnel-shaped distribution called Neal's Funnel [64]:

$$p(\mathbf{X}|\mathbf{Y}) = \mathcal{N}(\mathbf{Y}|0, 3)) \times \prod_{j=0}^{M-1} \mathcal{N}(\mathbf{X}^j|0, \exp(0.5\mathbf{Y})) \tag{2.39}$$

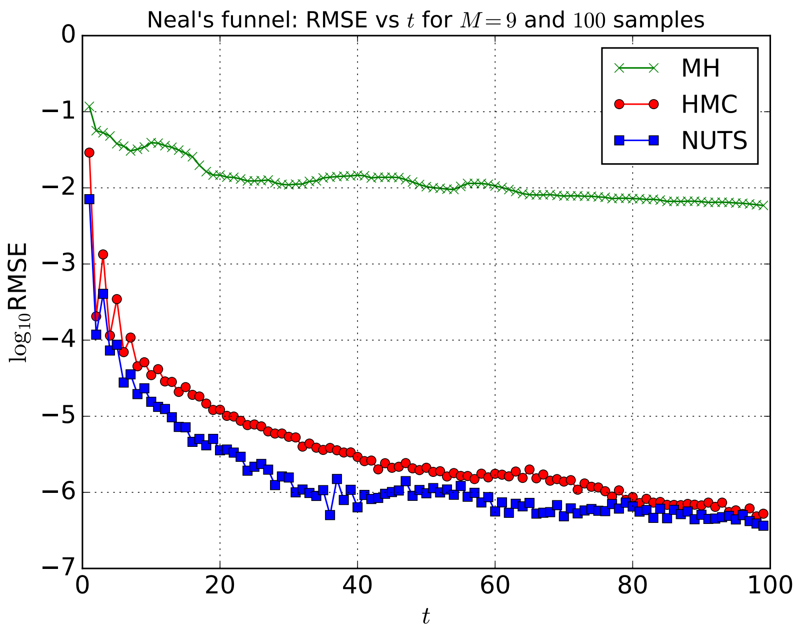
where $\mathbf{Y}$ is a one-dimensional data and the true mean for all funnels is 0.

Figure 2.4a shows the RMSE in $\log_{10}$ scale for all three MCMC methods and for $M = 9$. As we can see, Metropolis-Hastings converges more slowly than HMC and NUTS and does not achieve the same accuracy. This is because of the curse of dimensionality and because Metropolis-Hastings usually struggles with complicated targets such

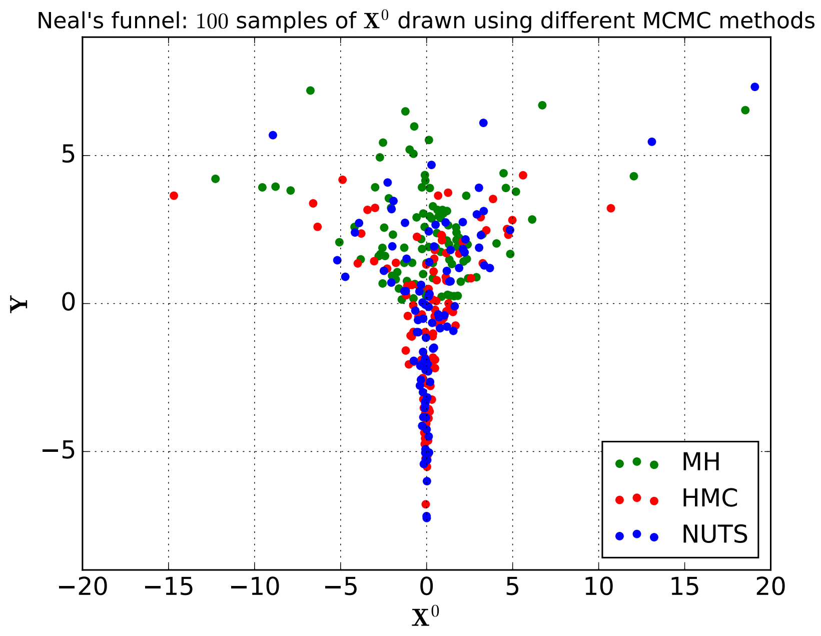
Algorithm 8 No-U-Turn Sampler (NUTS)

```
Input: $T_{NUTS}$, $\Delta h$, $p(\mathbf{X}|\mathbf{Y})$, $\mathbf{M}$, $\tau$
Output: $\mathbf{x}$
1: $\mathbf{x}_0 \sim q_0(\mathbf{x}_0)$, Draw the initial sample from the initial proposal
2: for $t \leftarrow 1; t \leq T_{HMC}; t \leftarrow t+1$ do
3:     $\mathbf{v} \sim \mathcal{N}(\mathbf{0}, \mathbf{M})$, draw a random momentum
4:     $u \sim \texttt{Uniform}[0, p(\mathbf{x}_{t-1}, \mathbf{v})]$, draw a slice variable
5:     $\mathbf{x}_+^* \leftarrow \mathbf{x}_{t-1}, \mathbf{x}_-^* \leftarrow \mathbf{x}_{t-1}, \mathbf{v}_+^* \leftarrow \mathbf{v}, \mathbf{v}_-^* \leftarrow \mathbf{v}, n \leftarrow 1, k \leftarrow 0, loop \leftarrow 1$
6:     while $loop = 1$ do
7:         $d^{'} \sim \texttt{Uniform}[\text{Heads}, \text{Tails}]$, flip an unbiased coin
8:         if $d^{'} = \text{Heads}$ then
9:             $\cdot, \cdot, \mathbf{x}_+^*, \mathbf{v}_+^*, \mathbf{x}^*, n^{'}, loop^{'} \leftarrow \texttt{Tree}(\mathbf{x}_+^*, \mathbf{v}_+^*), u, p(\mathbf{X}|\mathbf{Y}), \Delta h, k, 1)$
10:        else
11:            $\mathbf{x}_-^*, \mathbf{v}_-^*, \cdot, \cdot, \mathbf{x}^*, n^{'}, loop^{'} \leftarrow \texttt{Tree}(\mathbf{x}_-^*, \mathbf{v}_-^*), u, p(\mathbf{X}|\mathbf{Y}), \Delta h, k, -1)$
12:        end if
13:        if $loop^{'} = 1$ then
14:            if $\min(1, \frac{n^{'}}{n}) < \texttt{Uniform}[0,1]$ then
15:                $\mathbf{x}_t = \mathbf{x}^*$, the proposed sample is accepted
16:            else
17:                $\mathbf{x}_t = \mathbf{x}_{t-1}$, the proposed sample is rejected
18:            end if
19:        end if
20:        $loop \leftarrow loop^{'}\mathbb{I}((\mathbf{x}_+^* - \mathbf{x}_-^*) \cdot \mathbf{v}_-^* \geq 0)\mathbb{I}((\mathbf{x}_+^* - \mathbf{x}_-^*) \cdot \mathbf{v}_+^* \geq 0), k \leftarrow k+1, n \leftarrow n+n^{'}$
21:    end while
22: end for
23: Remove $\mathbf{x}_{0:\tau-1}$, burn-in the first $\tau$ samples
1: function TREE($\mathbf{x}, \mathbf{v}, u, p(\mathbf{X}|\mathbf{Y}), \Delta h, k, d$)
2:     if $j = 0$ then
3:         $\mathbf{x}^*, \mathbf{v}^* \leftarrow \texttt{Leapfrog}(\mathbf{x}_{t-1}, \mathbf{v}, 1, \Delta h, \mathbf{M}, p(\mathbf{X}|\mathbf{Y}), d)$, see Algorithm 6
4:         $n^{'} \leftarrow \mathbb{I}(u \in [0, p(\mathbf{X}, \mathbf{V})]), loop^{'} \leftarrow \mathbb{I}(p(\mathbf{X}, \mathbf{V} \geq \log u - 1000)$, see [44]
5:         return $\mathbf{x}^*, \mathbf{v}^*, \mathbf{x}^*, \mathbf{v}^*, \mathbf{x}^*, n^{'}, loop^{'}$
6:     else
7:         $\mathbf{x}_-, \mathbf{v}_-, \mathbf{x}_+, \mathbf{v}_+, \mathbf{x}^*, n^{'}, loop^{'} \leftarrow \texttt{Tree}(\mathbf{x}_-, \mathbf{v}_-, u, p(\mathbf{X}|\mathbf{Y}), \Delta h, k-1, d)$
8:         if $loop^{'} = 1$ then
9:             if $d = 1$ then
10:                $\cdot, \cdot, \mathbf{x}_+, \mathbf{v}_+, \mathbf{x}^{'}, n^{"}, loop^{"} \leftarrow \texttt{Tree}(\mathbf{x}_+^*, \mathbf{v}_+^*), u, p(\mathbf{X}|\mathbf{Y}), \Delta h, k-1, -1)$
11:            else
12:                $\mathbf{x}_-, \mathbf{v}_-, \cdot, \cdot, \mathbf{x}^{'}, n^{"}, loop^{"} \leftarrow \texttt{Tree}(\mathbf{x}_+^*, \mathbf{v}_+^*), u, p(\mathbf{X}|\mathbf{Y}), \Delta h, k-1, 1)$
13:            end if
14:            if $\min(1, \frac{n^{"}}{n^{'}+n^{"}}) < \texttt{Uniform}[0,1]$ then
15:                $\mathbf{x}^* = \mathbf{x}^{'}$, the proposed sample is accepted
16:            end if
17:            $loop^{'} \leftarrow loop^{"}\mathbb{I}((\mathbf{x}_+^* - \mathbf{x}_-^*) \cdot \mathbf{v}_-^* \geq 0)\mathbb{I}((\mathbf{x}_+^* - \mathbf{x}_-^*) \cdot \mathbf{v}_+^* \geq 0), n^{'} \leftarrow n^{'} + n^{"}$
18:        end if
19:        return $\mathbf{x}_-, \mathbf{v}_-, \mathbf{x}_+, \mathbf{v}_+, \mathbf{x}^*, n^{'}, loop^{'}$
20:    end if
21: end function
```

as (2.39). This can also be observed in Figure 2.4b which shows the samples of $\mathbf{X}^0$ (the other dimensions have the same shape and, therefore, are left out for brevity) that have been drawn using Metropolis-Hastings, HMC and NUTS. As we can see, Metropolis-Hastings struggles to explore the spout of the funnel, and hence most the samples are concentrated in the top region. In Chapter 6, the advantages that HMC and NUTS offer versus Metropolis-Hastings are applied to enhance the performance of SMC methods which also suffer from the curse of dimensionality.



(A) RMSE vs mean values



(B) Samples for $\mathbf{X}^0$

FIGURE 2.4: Neal's funnel example

Chapter 3

Parallelising Particle Filters with Deterministic Runtime on Distributed Memory Systems

3.1 Introduction

The (dynamic or static) models which SMC methods are commonly applied to are constantly being improved in several ways, such as adding more details and equations or using more sophisticated numerical solvers or typically both. Therefore, research is currently active on making SMC methods more accurate as well. Several solutions have already been proposed, such as using better proposal distributions [63, 79], better importance weight evaluations [10, 76, 101], or simply employing more particles [36, 52, 53] or any combination of them. These solutions share the same side effect: the run-time substantially increases and, most often, it does to the point that real-time applications of SMC methods get out of reach, due to critical measurement stream rate. Parallel computing is the typical solution address this problem.

SMC methods are often claimed to be easily parallelisable since the particles are drawn and weighted independently of each other. Although the IS step is indeed embarrassingly parallel, at some point it becomes necessary to perform resampling in order to correct degeneracy, as shown in Section 2.1.2. Resampling is however hard to parallelise globally, which means by using a parallelisation strategy on multiple cores that leads to the same outcome of sequential resampling. This is because of the difficulties in parallelising the constituent redistribute step, whose sequential textbook implementation S-R (see Algorithm 2) achieves $O(N)$ time complexity as (2.17) holds.

Several parallel redistributions have been proposed. A state-of-the-art parallelisable algorithm is presented in [85], has $O((\log_2 N)^2)$ time complexity and is implemented in the Big data MapReduce context. However, portability of this algorithm is difficult to achieve, especially on Distributed Memory Architectures (DMAs). The goal of this chapter is to reformulate this algorithm for DMAs by using MPI. This chapter demonstrates that the derived algorithm can be almost four times as fast as existing alternative MPI solutions for up to 256 cores. It also proves that the same algorithm can be reformulated to achieve further speed-up.

In doing so, the rest of this chapter is organised as follows: Section 3.2 provides a literature review of parallel redistribution strategies for DMAs. Section 3.3 illustrates how to implement on MPI all components of SMC methods. Section 3.4 describes how to port the redistribute in [85] to MPI; results for redistribution and an exemplary SMC method are also provided. Section 3.5 shows how to improve the derived redistribution and proves it by repeating the experiments in Section 3.4. Section 3.6 draws the conclusions of this chapter. The reader is recommended to consult Appendix A for details about DMAs and MPI.

3.2 Literature Review on Redistribute for DMAs

Although Algorithm 2 has a very low constant time and is very fast on a single core, it is also impossible to parallelise in embarrassingly parallel fashion, which means by equally dividing the iteration space across the parallel cores. This is due to the fact that the workload solely depends on the contents of **ncopies**, which is totally run-time dependent. As such, the workload can become extremely unbalanced depending on the elements of **ncopies**. On DMAs, parallelisation is further complicated by the fact that the memory spaces are partitioned and the cores cannot directly access the other cores' memory. Although all-to-all communication routines (e.g. `MPI_Alltoall` on MPI) can be used to provide easier data access across partitions, the time complexity would be downgraded to $O(N \log_2 N)$ for $P = N$ cores [66]. For obvious reasons, this parallel redistribute approach has never been attempted.

To bypass the need of parallelising resampling, one could use multiple PFs in parallel. This approach has already been explored several times, especially in multi-object tracking or economics applications [27, 38, 59, 82]. However, it has also shown three fundamental problems. The first one is that Multi-PFs typically do not provide both strong scaling and good accuracy, but they force the user to choose between one or the other. Since the PFs in a Multi-PF implementation run local resampling independently, it is strongly recommended to keep the number of particles per PF constant to N or at least establish a lower bound for that number [26, 56]. If this constraint is not satisfied, i.e. if the particles per PF progressively scale up as $O(\frac{N}{P})$, this approach asymptotically converges to a SIS filter which is known to be unusable, as shown in [25, 30]. Therefore, the run-time for two or more filters does not scale with respect to a single PF, but instead it slows down due to the increasing communication to combine the local estimates of all PFs. The second and third inconveniences are respectively about flexibility and applicability, and are both caused by the inherent model-dependent nature of this approach. In the cases which Multi-PFs can be used in, the filters need a model-dependent pre-tuning phase to divide the state space across the filters. This may be challenging depending on the model, and hence may strongly affect the performance [26]. In other cases, this approach is not feasible, for example when the posteriors of the marginalised states are multi-modal [26]. Because of these issues, Multi-PF approaches will not be taken into account in the rest of this thesis. Here it is specified that Multi-PFs can applied both on DMAs and SMAs with little to no parallelisation differences. This thesis now provides a description of some of the most recent and relevant parallel redistributions which have been specifically designed for DMAs.

Three DMA solutions (along with mixed versions of them) are presented in [12, 13] for Field Programmable Gate Array (FPGA): Centralised Resampling, Resampling with Proportional Allocation (RPA), and Resampling with Non-proportional Allocation (RNA). Although these approaches present substantial differences, they also share a similar core network: a central-unit controls a subset of the particles, but also acts as a job-scheduler, and hence is in charge of coordinating the communication between the other cores whose only duties are to duplicate, send and receive the remaining particles. Therefore, central-unit will be the technical name used in this work to generically refer to these strategies.

In Centralised Resampling, the central-unit gathers the particles from all cores, performs resampling over the full dataset of particles, and then scatters the result back to the cores. The implementation requires all-to-one and one-to-all communicators, such as `MPI_Gather` and `MPI_Scatter` on MPI. Since resampling (and redistribute) is performed by the central-unit in $O(N)$ and the communication cost grows linearly with P [66], we can infer that the run-time and time complexity are deterministic and data-independent but bound to $O(N)$ for any number of cores. For the same reasons, we can infer that the space complexity is also deterministic, data-independent and equal to $O(N)$ and a global resampling output is guaranteed. Broadly speaking, this idea tries to maximise the overall speed-up of PF by taking advantage of the fast constant time of Algorithm 2 in comparison with IS and the other PF components. However, because the computational effort of resampling is centralised to a single core, it is possible to prove that the speed-up saturates for a relatively low Degree Of Parallelism (DOP), according to Amdahl's law [41]. Another more unlikely issue is that, in the extreme scenario where N is so large that the particles do not fit the memory of the central-unit, this approach is obviously unfeasible. Centralised Resampling is found in several referenced work apart from [12, 13]: in [8], it is alternated with a decentralised resampling (akin to RNA), and used to correct the accuracy loss that the decentralised scheme causes during the SMC iterations. In [21], Centralised Resampling is employed in a Multi-PF implementation with a view of compensating for the degeneracy caused by SIS. In [39], a variant of Centralised Resampling on MPI is used for augmented resampling and is performed in three phases. In the first one, the central-unit gathers the weights from all cores and scatters back the normalised weights. After that, local resampling is performed by each core. In the final phase, the cores are coupled pairwise in a binary-tree structure: here, they exchange a certain amount of particle copies proportional to the normalised weights, in order to achieve weight distribution.

In RPA, all cores (and not only the central-unit) are actively involved in creating the particle copies during redistribute. Therefore, the central-unit partially performs redistribution, but also decides the exact number of copies that the other cores must create, meaning that the output is the same as in Algorithm 2. This load-balancing decision, however, strongly depends on the input unbalance level over the cores. As such, RPA is strongly data-dependent and so is its run-time during the SMC iterations. The best scenario is when the input workload happens to be balanced already, which costs no extra communication between the cores. However, the worst-case occurs when one element in **ncopies** equals N and all the others are equal to 0: this translates to a single core having to communicate the copies in excess to the other cores (as in Centralised Resampling), providing no speed-up for any value of P. In the average case, the speed-up may also saturate rapidly as shown in [83]. Apart from [12, 13], another application of RPA can be found in [100] on MPI.

RNA has been developed with a view of reducing the communication cost in redistribution. In doing so, the computational load of running the central-unit here is lighter than in RPA and Centralised Resampling. However, this comes with a risk of causing a statistical accuracy loss. The first step here is indeed local resampling, as in Multi-PF approaches. This leads to an uneven distribution of the weights. Therefore, the particles are cyclically exchanged between neighbor cores on a ring topology network, and the communication goes on until the weights are evenly distributed. The number of particles to send per message can be established deterministically at compile-time, and is typically chosen to be about $10 - 50\%$ of the number of particles per core. However, since it is hard to determine an optimal value for the number of particles to send per message, there is a high risk of experiencing redundant communication, especially for large N, which may affect the speed-up [25, 83]. Also, since a local resampling step is performed, the output may not be the same as Algorithm 2, which could compromise the overall accuracy. [1] is another referenced work where RNA is employed (besides [12, 13]), while in [8] a variant of RNA is alternated with Centralised Resampling, as previously mentioned.

In summary, we can say that central-unit strategies may have accuracy or scalability issue, especially for highly unbalanced workload, large N or DOP, as shown in [25, 83]. This thesis is then mostly focused on parallel *fully-balanced* redistribute solutions which are defined as follows.

Definition 3.1. A fully-balanced redistribute meets the following requests:

- all cores perform the same pre-agreed tasks (i.e. no central-unit(s) involved) to balance the workload (i.e. the number of particles per core) evenly;
- the achieved time complexity for P cores has to be faster than the time complexity for one core in order to guarantee strong scaling performance;

- the redistribution of the particles is global to ensure the same output of its sequential version and no speed-accuracy trade-off is made when P increases.

Another preferable request for fully-balanced redistribute is to have deterministic run-time which is crucial in real-time domains. When it comes to redistribution on DMAs, another preferable request is to have deterministic and scalable space complexity.

In [58], it has been shown that redistribute can be parallelised in a fully-balanced fashion by using a divide-and-conquer approach. By using a balanced binary tree, this algorithm recursively sorts and splits each node into two leaves, leaving half of the number of copies on each side. This algorithm was implemented in C for Graphics. In order to sort the particles, [58] employs Bitonic Sort, a deterministic, comparison-based parallel sorting algorithm which takes $O((\log_2 N)^2)$ comparisons for $P = N$ parallel cores [96]. Since Bitonic Sort is called $\log_2 N$ times in [58], the achieved time complexity is $O((\log_2 N)^3)$ (not $O((\log_2 N)^2)$ as claimed in [58]). In [25], the idea of using sort recursively has been applied in a dynamic scheduler for RPA/RNA and implemented on Message Passing Interface (MPI). In [85], the time complexity has been reduced to $O((\log_2 N)^2)$ by proving that Bitonic Sort is only needed once. This algorithm has been implemented on MapReduce and, although it was significantly better than the algorithm in [58], its run-time for 512 cores was at best 20 times worse than S-R [85]. This is because the communication network in this MapReduce implementation is built on a high level of abstraction and the consequent data movement is highly inefficient [85]. The goals in this chapter and Chapter 4 are to first propose a better framework for the algorithm [85] and then redesign the same to improve the performance.

Ref.	**Type**	**Parallel computing platform**	**Rando-mised**	**Global**	**Worst-case Time Complexity**	**Space Complexity**
[26]	Multi-PF	Simulation	No	No	$O(N)$	$O(N)$
[27]	Multi-PF	Simulation	No	No	$O(N)$	$O(N)$
[59]	Multi-PF	Simulation	No	No	$O(N)$	$O(N)$
[38]	Multi-PF	Simulation	No	No	$O(N)$	$O(N)$
[82]	Multi-PF	MPI	No	No	$O(N)$	$O(N)$
[12]	Central	FPGA	No	Yes	$O(N)$	$O(N)$
[8]	Central+RNA	Simulation	No	No	$O(N)$	$O(N)$
[39]	Central	MPI	No	Yes	$O(N)$	$O(N)$
[21]	Central	Simulation	No	Yes	$O(N)$	$O(N)$
[12]	RPA	FPGA	Yes	Yes	$O(N)$	$O(1)$
[13]	RPA	FPGA	Yes	Yes	$O(N)$	$O(1)$
[100]	RPA	MPI	Yes	Yes	$O(N)$	$O(1)$
[83]	RPA	Simulation	Yes	Yes	$O(N)$	$O(1)$
[12]	RNA	FPGA	No	No	$O(N)$	$O(1)$
[13]	RNA	FPGA	No	No	$O(N)$	$O(1)$
[83]	RNA	Simulation	No	No	$O(N)$	$O(1)$
[1]	RNA	Simulation	No	No	$O(N)$	$O(1)$
[58]	Fully-balanced	C-Graphics	No	Yes	$O((\log_2 N)^3)$	$O(1)$
[85]	Fully-balanced	MapReduce	No	Yes	$O((\log_2 N)^2)$	$O(1)$

TABLE 3.1: Literature review on redistribute for DMAs.

3.3 SMC Methods on DMAs

This section describes how to implement an SMC method on DMAs by using MPI. In doing so, the following subsections describe the most suitable parallelisation strategy for each PF component, illustrate their implementation on MPI along with scalability results. All reported run-times and speed-ups in this Chapter are medians of 20 runs for the same N, P pair. Table 3.2 provides hardware details about the cluster that has been used for this thesis. The reader is referred to Appendix A for more details on MPI.

TABLE 3.2: Details of the clusters.

Name	**Barkla**
OS	CentOS Linux 7
Number of Nodes	8
Cores per node	40
CPU	2 Xeon Gold 6138
RAM	384 GB
Clock	2.2 GHz
Cache L2	20 MB
MPI Version	OpenMPI-1.10.1
Interconnect	Infiniband 100 Gbps
Job scheduler	Slurm

3.3.1 Embarrassingly Parallel

`Reset`, `Initialise`, Equation (2.18) and all variants of IS, such as (2.12) and (2.13) for the PF, (2.22) and (2.23) for the SMC sampler, and (2.26) and (2.28) for Fixed-Lag SMC are embarrassingly parallel, since the computation of each i-th element is independent of the others. Therefore, a correct parallelisation consists of equally dividing the iteration space across the cores. This means that each core always owns $n = \frac{N}{P}$ elements of every array involved.

On DMAs, it also important to decide how to distribute the elements of all arrays across the cores. Several partitioning strategies can be used on MPI. The most intuitive one is to assign the array indexes to the cores in increasing order. More precisely, given a certain N, P pair, the i-th element (where $0 \leq i \leq N-1$) will always belong to the same core with MPI rank $p = int(i/n)$. Therefore, it is trivial to infer that embarrassingly parallel equations achieve $O(\frac{N}{P})$ time and space complexities, which converge to $O(1)$ when $P = N$. The same partitioning strategy is used in all parallel algorithms that are described in this thesis, since a $O(1)$ space complexity is one of the requirements for implementing a fully-balanced parallel redistribute.

Algorithms 9 and 10 describe the pseudo-code for `Reset` and the IS step in Algorithm 1. The implementation for `Initialise` and other variants of IS are omitted for brevity, since they are equivalent to Algorithm 9.

Algorithm 9 Importance Sampling (IS)

```
Input: $\mathbf{x}_{t-1}$, $\mathbf{w}_{t-1}$, $\mathbf{Y}_t$, $N$, $P$
Output: $\mathbf{x}_t$, $\mathbf{w}_t$
1: $n \leftarrow \frac{N}{P}$
2: for $i \leftarrow 0$; $i < n$; $i \leftarrow i + 1$ do
3:     $\mathbf{x}_t^i \sim q(\mathbf{x}_t^i | \mathbf{x}_{t-1}^i, \mathbf{Y}_t)$
4:     $\mathbf{w}_t^i \leftarrow \mathbf{w}_{t-1}^i \frac{p(\mathbf{Y}_t|\mathbf{x}_t^i)p(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i)}{q(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i, \mathbf{Y}_t)}$
5: end for
```

Algorithm 10 Reset

```
Input: $\mathbf{w}$, $N$, $P$
Output: $\mathbf{w}$
1: $n \leftarrow \frac{N}{P}$
2: for $i \leftarrow 0$; $i < n$; $i \leftarrow i + 1$ do
3:     $\mathbf{w}^i \leftarrow \frac{1}{N}$
4: end for
```

These algorithms show that embarrassingly parallel computations require no communication between the cores. As we can see in Figure 3.1, the speed-up vs a sequential implementation of `Reset` becomes closer to P for high values of N, due to the absence of data movement and the increasing volume of computation. Here, the speed-up of IS is purposely omitted as IS is a model-dependent task and hence the results may vary depending on the given model. Information on the scalability of IS will be given in the case studies throughout the rest of the thesis.

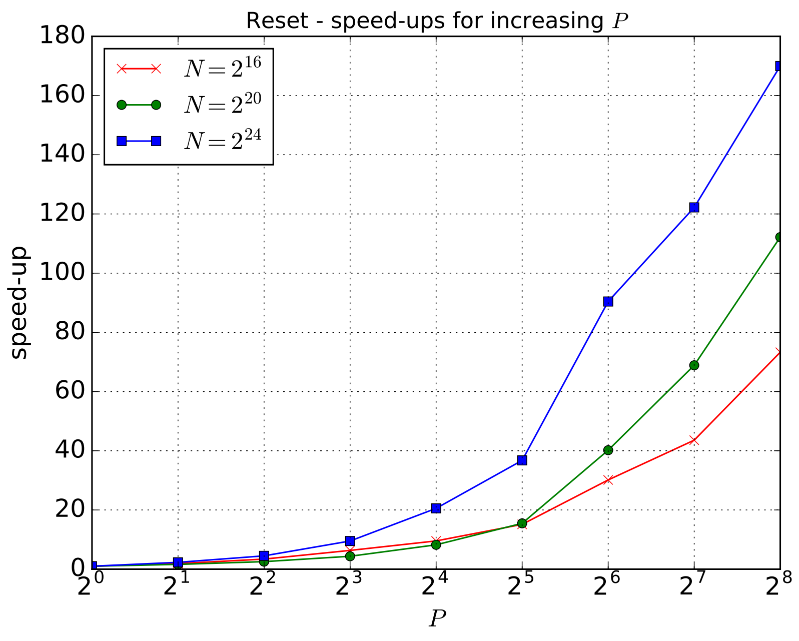


FIGURE 3.1: Reset - speed-up for up to $N = 2^{24}$ and $P = 256$

3.3.2 Reduction

The Sum in (2.14), (2.16), and for the estimates (2.15) and (2.19) can be easily parallelized by using reduction, a divide-and-conquer operation which scales logarithmically with P, more precisely as $O(\frac{N}{P} + \log_2 P)$. The mathematical intuition behind reduction is that the Sum of a N-element array can be computed by adding the partial Sum of the first $\frac{N}{2}$ elements of the array to the partial Sum of its other half. However, these two partial Sums can be split the same way themselves by taking the Sums of their two halves, i.e. the quarters of the original array. By repeating this reduction process recursively, we obtain:

$$\begin{aligned}
\sum_{i=0}^{N-1} \mathbf{w}^i &= \sum_{i=0}^{\frac{N}{2}-1} \mathbf{w}^i + \sum_{i=\frac{N}{2}}^{N-1} \mathbf{w}^i \\
&= \sum_{i=0}^{\frac{N}{4}-1} \mathbf{w}^i + \sum_{i=\frac{N}{4}}^{\frac{N}{2}-1} \mathbf{w}^i + \sum_{i=\frac{N}{2}}^{\frac{3N}{4}-1} \mathbf{w}^i + \sum_{i=\frac{3N}{4}}^{N-1} \mathbf{w}^i \\
&\vdots \\
&= (\mathbf{w}^0 + \mathbf{w}^1) + (\mathbf{w}^2 + \mathbf{w}^3) + \cdots + (\mathbf{w}^{N-3} + \mathbf{w}^{N-2}) + (\mathbf{w}^{N-2} + \mathbf{w}^{N-1}) \\
&= \mathbf{w}^0 + \mathbf{w}^1 + \mathbf{w}^2 + \mathbf{w}^3 + \cdots + \mathbf{w}^{N-3} + \mathbf{w}^{N-2} + \mathbf{w}^{N-2} + \mathbf{w}^{N-1}
\end{aligned}$$

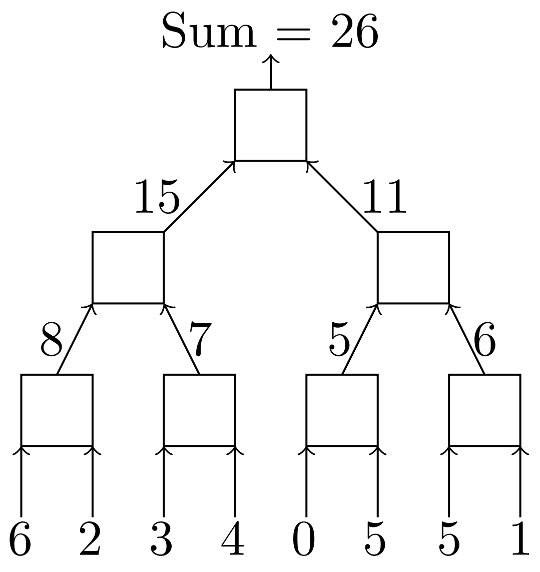


FIGURE 3.2: Reduction - binary tree

This way it is possible to parallelise Sum in $O(\log_2 N)$ stages by using a binary-tree structure. During each stage, the parallel cores are divided into groups (or nodes). The cores within the same node are interconnected pairwise such that each core sends to its partner the partial Sum from the previous stage. Figure 3.2 illustrates the binary-tree structure for reduction, where the squares represent core nodes and the arrows represent communication of partial Sums.

On MPI, reduction can be computed by calling `MPI_reduce` or `MPI_Allreduce` on the variable to reduce, after an $O(\frac{N}{P})$ sequential computation of the operation to reduce. Figure 3.3 illustrates a possible example for $N = 16$ and $P = 4$. It is also worth mentioning that reduction can be used the same way to parallelise any other elementary operation such as Product, Max or Min.

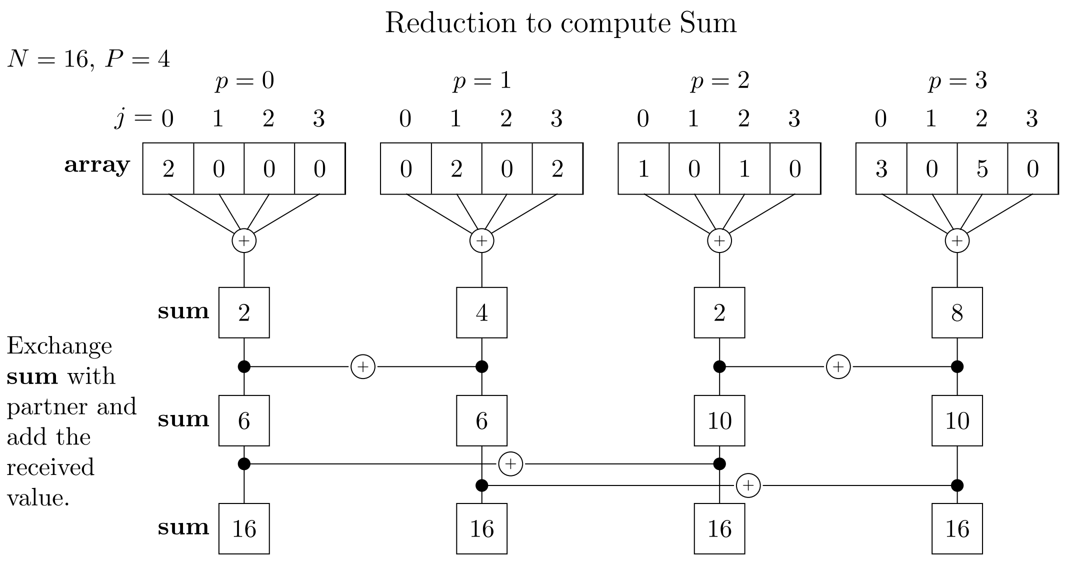


FIGURE 3.3: MPI Reduction - Example for $N = 16$ and $P = 4$

Algorithms 11, 12 and 13 describe MPI pseudo-codes for `Normalise`, `ESS` and `Mean`. The MPI implementation of (2.15) is omitted for brevity as it is equivalent to `Mean`.

Algorithm 11 Normalise

Input: $\mathbf{w}$, N, P
Output: $\tilde{\mathbf{w}}$

```
1: $n \leftarrow \frac{N}{P}$
2: $local\_sum \leftarrow 0$
3: for $i \leftarrow 0$; $i < n$; $i \leftarrow i + 1$ do
4:     $local\_sum \leftarrow local\_sum + \mathbf{w}^i$
5: end for
6: MPI_Allreduce($local\_sum, sum, 1$, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD)
7: for $i \leftarrow 0$; $i < n$; $i \leftarrow i + 1$ do
8:     $\tilde{\mathbf{w}}^i \leftarrow \frac{\mathbf{w}^i}{sum}$
9: end for
```

Figure 3.4 illustrates the scalability results for `Normalise`. Simular results can be found for `ESS` and `Mean` but are not provided for brevity. Here, as in Figure 3.1, we can also observe that the speed-up becomes more linear for high values of N, due to the increasing computation-vs-communication granularity. However, for lower N and high DOP, the granularity gets fine and the speed-up saturates.

Algorithm 12 Effective Sample Size (ESS)

```
Input: $\tilde{\mathbf{w}}$, $N$, $P$,
Output: $N_{eff}$
1: $n \leftarrow \frac{N}{P}$
2: $local\_sum \leftarrow 0$
3: for $i \leftarrow 0$; $i < n$; $i \leftarrow i + 1$ do
4:     $local\_sum \leftarrow local\_sum + (\tilde{\mathbf{w}}^i)^2$
5: end for
6: MPI_Allreduce($local\_sum, sum, 1$, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD)
7: $N_{eff} \leftarrow \frac{1}{sum}$
```

Algorithm 13 Mean

```
Input: $\mathbf{x}$, $N$, $P$, $M$
Output: $\xi$
1: $n \leftarrow \frac{N}{P}$
2: local_sum ← 0, local_sum and sum are in bold here because $\mathbf{x}^i \in \mathbb{R}^M$
3: for $i \leftarrow 0$; $i < n$; $i \leftarrow i + 1$ do
4:     local_sum ← local_sum + $\mathbf{x}^i$
5: end for
6: MPI_Allreduce(local_sum, sum, $M$, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD)
7: $\xi \leftarrow \frac{\mathbf{sum}}{N}$
```

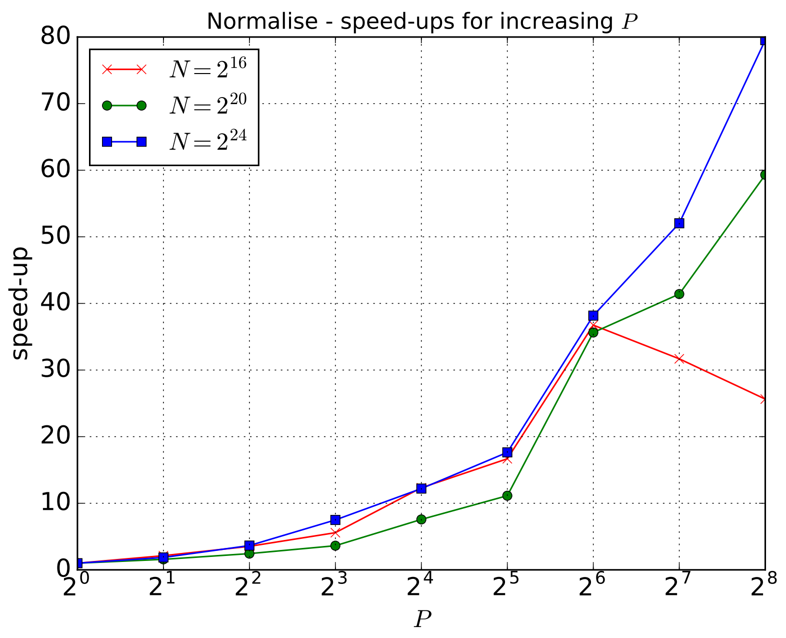


FIGURE 3.4: Normalise - speed-up for up to $N = 2^{24}$ and $P = 256$

3.3.3 Cumulative Sum

The CDF of the weights requires Cumulative Sum, which can also be found in the literature under the name of Prefix Sum or Scan. This operation can be computed either in inclusive or exclusive form. More precisely, given a N-element input vector

array, the i-th element of the output vector **csum** is:

$$\mathbf{csum}^i = \mathbf{csum}^{i-1} + \mathbf{array}^i \tag{3.1}$$

where $\mathbf{csum}^0 = \mathbf{array}^0$ if Cumulative Sum is computed in inclusive form, or $\mathbf{csum}^0 = 0$ if Cumulative Sum is computed in exclusive form.

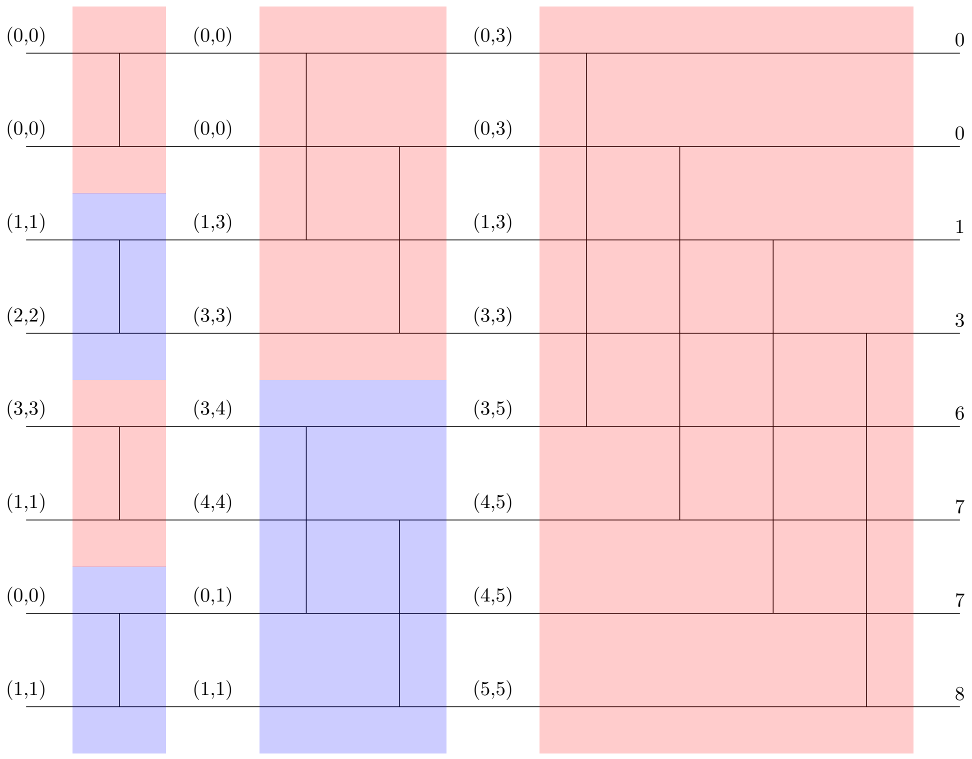
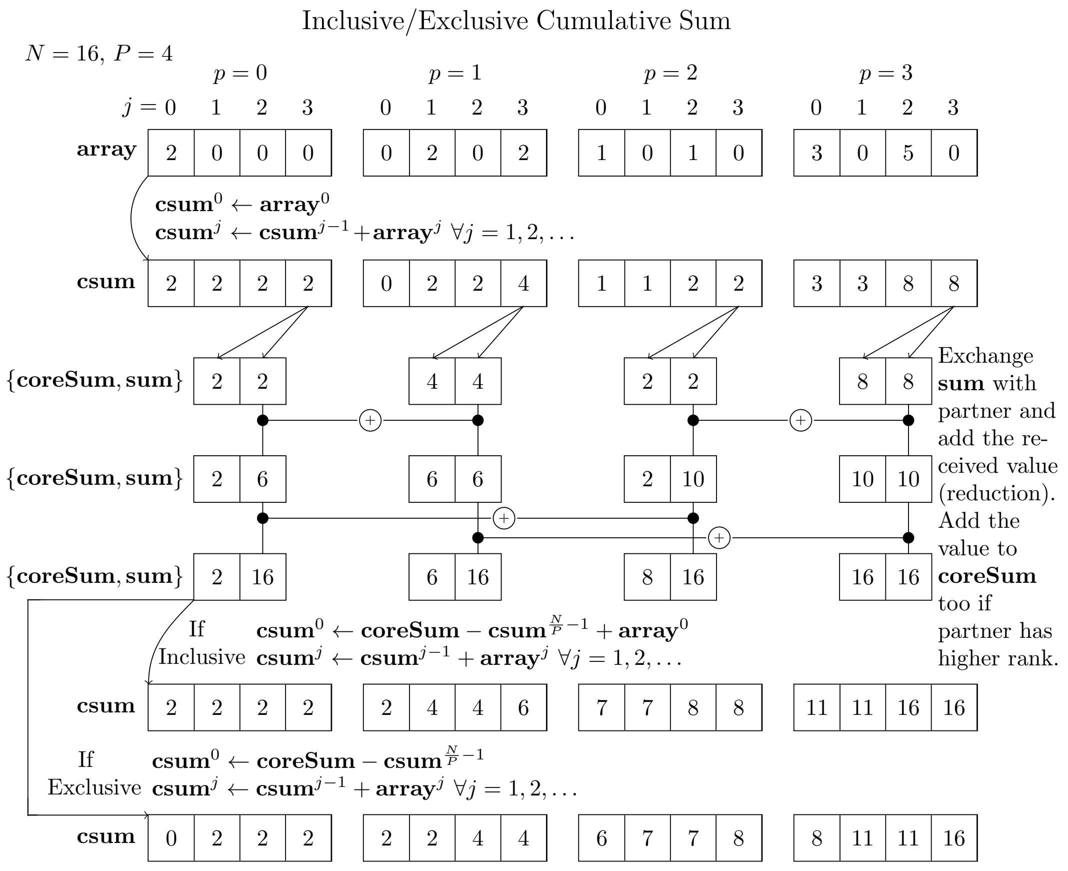


FIGURE 3.5: Parallel Cumulative Sum - binary tree structure

Parallel Cumulative Sum was first presented in [45] and then optimised in [19]; more recent versions can be found in [60, 81]. These implementations scale logarithmically, more precisely as $O(\frac{N}{P} + \log_2 P)$. The idea consists of using a binary-tree structure, akin to Figure 3.2. As they move to the top of the tree, the cores (uniquely identified by a rank) keep track of two variables: the node partial Sum and the node partial Cumulative Sum. At each stage, the cores within the same node are coupled pairwise and send the partial Sum to their partner. The partial Sum is updated at every stage just like in reduction. However, the partial Cumulative Sum is updated by adding the received value, only if the core's rank is higher than its partner's rank. Figure 3.5 illustrates an example for $P = 8$ cores: the coloured blocks represent core nodes and the exchange of partial Sums between partner cores is represented by the vertical arrows.

On MPI, parallel Cumulative Sum can be performed in inclusive form by calling `MPI_Scan` and in exclusive form by calling `MPI_Exscan` [66]. If $P < N$, two $O(\frac{N}{P})$ calls of (3.1) are required to initialise and finalise `MPI_Scan`/`MPI_Exscan`. Algorithm 14 illustrates a pseudo-code of parallel Cumulative Sum on MPI. An example for $N = 16$ elements and $P = 4$ MPI cores can be found in Figure 3.6.



Inclusive/Exclusive Cumulative Sum
$N = 16$, $P = 4$
$p = 0$ $p = 1$ $p = 2$ $p = 3$
$j = 0$ 1 2 3
array
$\mathbf{csum}^0 \leftarrow \mathbf{array}^0$
$\mathbf{csum}^j \leftarrow \mathbf{csum}^{j-1} + \mathbf{array}^j \ \forall j = 1, 2, \ldots$
csum
$\{\mathbf{coreSum}, \mathbf{sum}\}$
Exchange **sum** with partner and add the received value (reduction). Add the value to **coreSum** too if partner has higher rank.
If Inclusive $\mathbf{csum}^0 \leftarrow \mathbf{coreSum} - \mathbf{csum}^{\frac{N}{P}-1} + \mathbf{array}^0$
$\mathbf{csum}^j \leftarrow \mathbf{csum}^{j-1} + \mathbf{array}^j \ \forall j = 1, 2, \ldots$
If Exclusive $\mathbf{csum}^0 \leftarrow \mathbf{coreSum} - \mathbf{csum}^{\frac{N}{P}-1}$
$\mathbf{csum}^j \leftarrow \mathbf{csum}^{j-1} + \mathbf{array}^j \ \forall j = 1, 2, \ldots$

FIGURE 3.6: MPI Cumulative Sum - example for $N = 16$ and $P = 4$

Algorithm 15 illustrates a possible pseudo-code for MVR, the first task in Algorithms 1, 3 and 4 where parallel Cumulative Sum is required. Figures 3.7 and 3.8 report the scalability results for single calls of Algorithms 14 and 15 under the same testing conditions of Section 3.3.1 and 3.3.2. Once again, we can observe higher efficiency for larger N due to the increasing granularity.

3.3.4 $O((\log_2 N)^3)$ Fully-Balanced Redistribute

As discussed in Section 3.2, S-R is impossible to parallelise in element-wise fashion, i.e. by simply dividing equally the iteration space across the cores. This is because each **ncopies**^{i} randomly changes at every time step t as it may be equal to any integer number between 0 and N. Therefore, an element-wise parallelization could be extremely unbalanced. On DMAs, a deterministic parallelisation is even more problematic as the cores can only directly access their own private memory. In opposition to central-unit approaches such as RPA and RNA, fully-balanced strategies offer a parallelisation for global resampling with deterministic and scalable run-time and private memory space. This section describes how to implement on MPI the redistribute parallelisation in [58], the first fully-balanced redistribute that can be found in the literature.

Since the input workload in S-R is non-deterministic, the goal is to move the particles deterministically by using a divide-and-conquer routine which achieves a faster time

Algorithm 14 MPI Cumulative Sum

```
Input: $\mathbf{array}$, $N$, $P$, $type$, $form$, $comm$
Output: $\mathbf{csum}$
1: $n \leftarrow \frac{N}{P}$
2: $\mathbf{csum}^0 \leftarrow \mathbf{array}^0$
3: for $i \leftarrow 1; i < n; i \leftarrow i + 1$ do
4:     $\mathbf{csum}^i \leftarrow \mathbf{csum}^{i-1} + \mathbf{array}^i$
5: end for
6: $\texttt{MPI\_Scan}(\mathbf{csum}^{n-1}, \mathbf{coreSum}, 1, type, \texttt{MPI\_SUM}, comm)$
7: if $form == \texttt{Inclusive}$ then
8:     $\mathbf{csum}^0 \leftarrow \mathbf{coreSum} - \mathbf{csum}^{n-1} + \mathbf{array}^0$
9: else
10:    $\mathbf{csum}^0 \leftarrow \mathbf{coreSum} - \mathbf{csum}^{n-1}$
11: end if
12: for $i \leftarrow 1; i < n; i \leftarrow i + 1$ do
13:    $\mathbf{csum}^i \leftarrow \mathbf{csum}^{i-1} + \mathbf{array}^i$
14: end for
```

Algorithm 15 Minimum Variance Resampling (MVR)

```
Input: $\tilde{\mathbf{w}}$, $N$, $P$, $p$
Output: $\mathbf{ncopies}$
1: $n \leftarrow \frac{N}{P}$
2: $comm \leftarrow \texttt{MPI\_COMM\_WORLD}$
3: if $p == 0$ then
4:     $\mathbf{cdf}^0 \leftarrow 0$, because $\mathbf{cdf} \in \mathbb{R}^{N+1}$
5: end if
6: if $p == 0$ then
7:     $\mathbf{cdf}^{1:n} \leftarrow \texttt{MPI\_Cumulative\_Sum}(\tilde{\mathbf{w}}, N, P, \texttt{MPI\_DOUBLE}, \texttt{Inclusive}, comm)$
8: else
9:     $\mathbf{cdf}^{1:n-1} \leftarrow \texttt{MPI\_Cumulative\_Sum}(\tilde{\mathbf{w}}, N, P, \texttt{MPI\_DOUBLE}, \texttt{Inclusive}, comm)$
10: end if
11: $u \sim \texttt{Uniform[0,1)}$
12: $\texttt{MPI\_Bcast}(u, 1, \texttt{MPI\_DOUBLE}, 0, comm)$, broadcast $u$ to other cores in $O(\log_2 P)$
13: for $i \leftarrow 0; i < n; i \leftarrow i + 1$ do
14:     $\mathbf{ncopies}^i \leftarrow \lceil \mathbf{cdf}^i + \tilde{\mathbf{w}}^i - u \rceil - \lceil \mathbf{cdf}^i - u \rceil$
15: end for
```

complexity than $O(N)$, the number of memory stores performed by Algorithm 2, as proven by (2.17). More precisely, the goal is to achieve perfect load-balancing across the MPI cores. This is equivalent to having:

$$\sum_{i=p\frac{N}{P}}^{(p+1)\frac{N}{P}-1} \mathbf{ncopies}^i = \frac{N}{P} \quad \forall p = 0, \dots, P-1 \tag{3.2}$$

which is essentially (2.17) applied locally. Once (3.2) is satisfied, the number of particles to duplicate owned by each MPI core is balanced. Hence, the cores can safely perform Algorithm 2 within their memory in $O(\frac{N}{P})$.

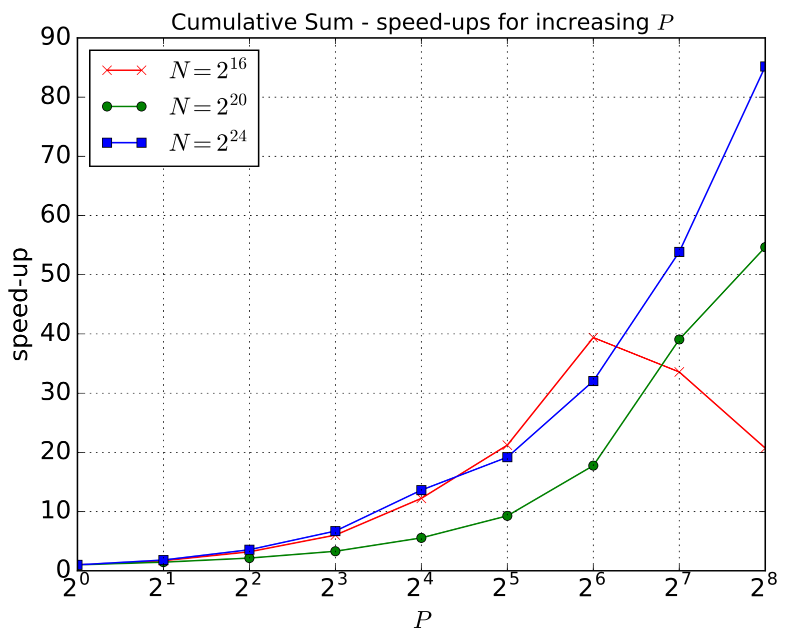


FIGURE 3.7: MPI Cumulative Sum - speed-up for up to $N = 2^{24}$ and $P = 256$

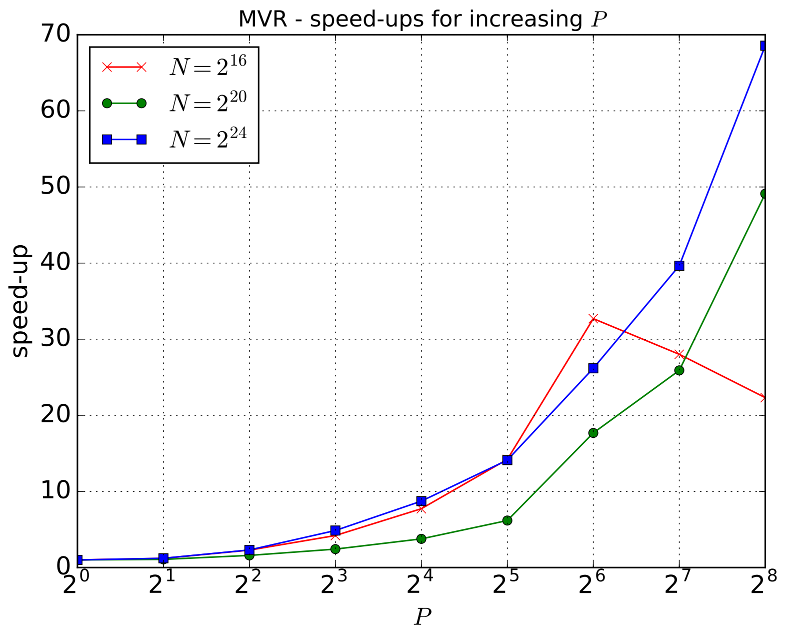


FIGURE 3.8: MVR - speed-up for up to $N = 2^{24}$ and $P = 256$

The divide-and-conquer strategy in [58] uses a top-down binary-tree structure. Starting from the root node, the key idea consists of sorting **ncopies** and moving **x** consequently at every stage of the binary tree. This way, the given randomised input pair **ncopies**, **x** is re-organised into a known, ordered input. After that, it is possible to split deterministically the father node into two balanced child nodes, each having half of the cores and half of particles to duplicate. To achieve that, the cores search for a particular index, called *pivot*, such that $\frac{N}{2}$ particles to copy can be counted on each side of *pivot*. In order to find *pivot*, parallel Cumulative Sum is performed over **ncopies** and then *pivot* is the first index such that the same Cumulative Sum is equal to or greater than $\frac{N}{2}$. Once *pivot* is identified, the cores are coupled pairwise as in the reduction operation, and send to their partner all the particles whose index $i \leq pivot$. Since the particles per node are halved at each stage, (3.2) is achieved by calling this routine recursively $\log_2 P$ times.

Apart from Cumulative Sum, which has been discussed already in the previous section, this parallel routine requires other two components: sort and pivot calculation, which are described in the two following sections.

3.3.4.1 Bitonic Sort

To sort **ncopies** and **x**, the redistribution in [58] uses Bitonic Sort, a fast comparison-based parallel sorting algorithm. Bitonic Sort was first presented in [9] and recently implemented on a cluster of graphics cards in [96]. This algorithm uses a divide-and-conquer approach to first divide the input sequence into a series of Bitonic sequences[1]. Then the Bitonic sequences are recursively merged together until the algorithm returns a single monotonic sorted sequence.

A possible sorting network which can be used is illustrated in Figure 3.9. Each horizontal wire represents a key, the vertical arrows connect the input keys for a comparison and the direction represents the order of the keys after the comparison has occurred. The coloured blocks represent the tree nodes (blue or red if the input keys must be sorted in increasing or decreasing order respectively).

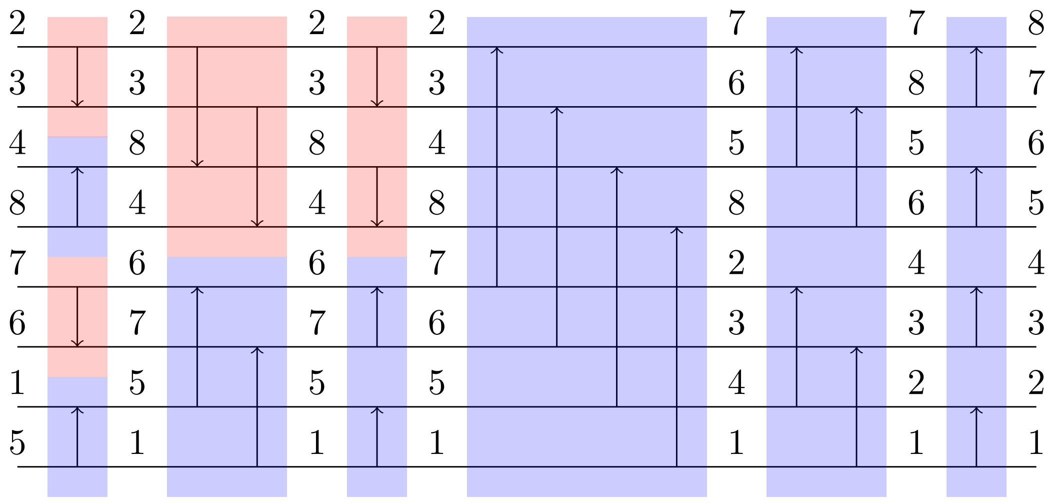


FIGURE 3.9: Bitonic Sort - sorting network

For the purposes of this thesis, a modified version of Bitonic Sort is needed, because while the keys in **ncopies** are sorted, the particles will move the same way. Algorithm 16

[1]A Bitonic sequence is a sequence of N keys in which the first $N/2$ keys are sorted in increasing order, while the last $N/2$ keys are sorted in decreasing order.

Algorithm 16 MPI Bitonic Sort

```
Input: x, ncopies, $N$, $P$, $p$, $comm$
Output: x, ncopies
 1: $n \leftarrow \frac{N}{P}$
 2: Allocate $s$, an MPI_Status variable
 3: Serial_Bitonic_Sort($\mathbf{x}, \mathbf{ncopies}, n$)
 4: for $i \leftarrow 2$; $i \leq P$; $i \leftarrow 2 \cdot i$ do
 5:     $up \leftarrow$ Direction($p, i$)
 6:     for $j \leftarrow 0$; $j < \log_2 i$; $j \leftarrow j + 1$ do
 7:         $par \leftarrow$ PartnerCalc($p, i, j$)
 8:         MPI_Sendrecv($\mathbf{ncopies}, n$, MPI_INT, $par, 0, \mathbf{tmp}, n$, MPI_INT, $par, 0, comm, s$)
 9:         MPI_Sendrecv($\mathbf{x}, n$, MPI_DOUBLE, $par, 0, \mathbf{tmpx}, n$, MPI_DOUBLE, $par, 0, comm, s$)
10:         Merge($\mathbf{x}, \mathbf{ncopies}, \mathbf{tmp}, \mathbf{tmpx}, up, p, par, n$)
11:     end for
12: end for
```

illustrates a possible pseudo-code to implement on MPI the sorting network in Figure 3.9. As can be observed from the algorithm, each MPI process starts by sorting the particles locally. The implementation of `Serial_Bitonic_Sort` is omitted for brevity as it resembles the two nested for loops in Algorithm 16, with the only difference of having steps 7, 8 and 9 switched with a for loop, which iteratively selects and compares two keys at the time. After local sort, the cores are organised into nodes in a bottom-up binary-tree structure. Each node is itself a top-down binary tree, where the cores are coupled pairwise. Therefore, at every stage of the top-down binary tree, each core selects a new partner to exchange the whole content of **ncopies** and **x** with. After that, `Merge`[2] is invoked locally in order to keep the lowest or highest keys, according to the sorting direction *up*. Since `Merge` takes $O(\frac{N}{P})$ comparisons and is invoked $(\log_2 P)^2$ times, we can infer infer that the achieved time complexity is equal to $O((\log_2 N)^2)$ for $P = N$ processors. For any $P \leq N$, Bitonic Sort performs

$$O\left(\frac{N}{P}\left(\log_2\left(\frac{N}{P}\right)\right)^2 + \frac{N}{P}\left(\log_2 P\right)^2\right) \tag{3.3}$$

comparisons, where the first term describes the number of steps to perform Bitonic Sort locally and the second term represents the data movement to merge the keys between the cores.

Figure 3.10 shows the speed-ups for Bitonic Sort, using the same values of N and P in Sections 3.3.1, 3.3.2 and 3.3.3. Once again, we can see that the speed-ups increase with N, but in this case the values are lower since the communication in Bitonic Sort is heavier than in reduction or Cumulative Sum.

[2]`Merge` is a $O(N)$ routine which takes two ordered N-element sequences and saves either the lowest or the highest N keys, depending on the direction of the arrow in Figure 3.9, the rank p or the partner's rank par. The implementation here is omitted for brevity but an exhaustive explanation can be found in [7].

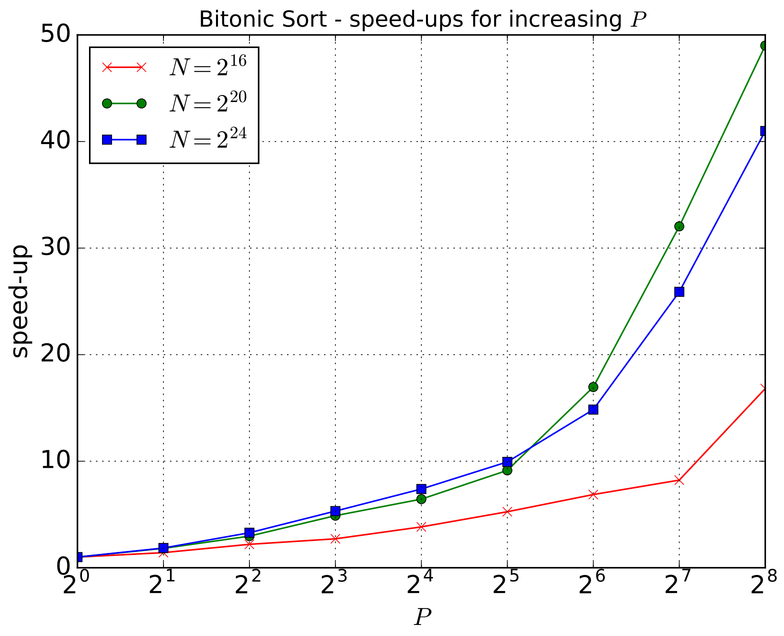


FIGURE 3.10: MPI Bitonic Sort - speed-up for up to $N = 2^{24}$ and $P = 256$

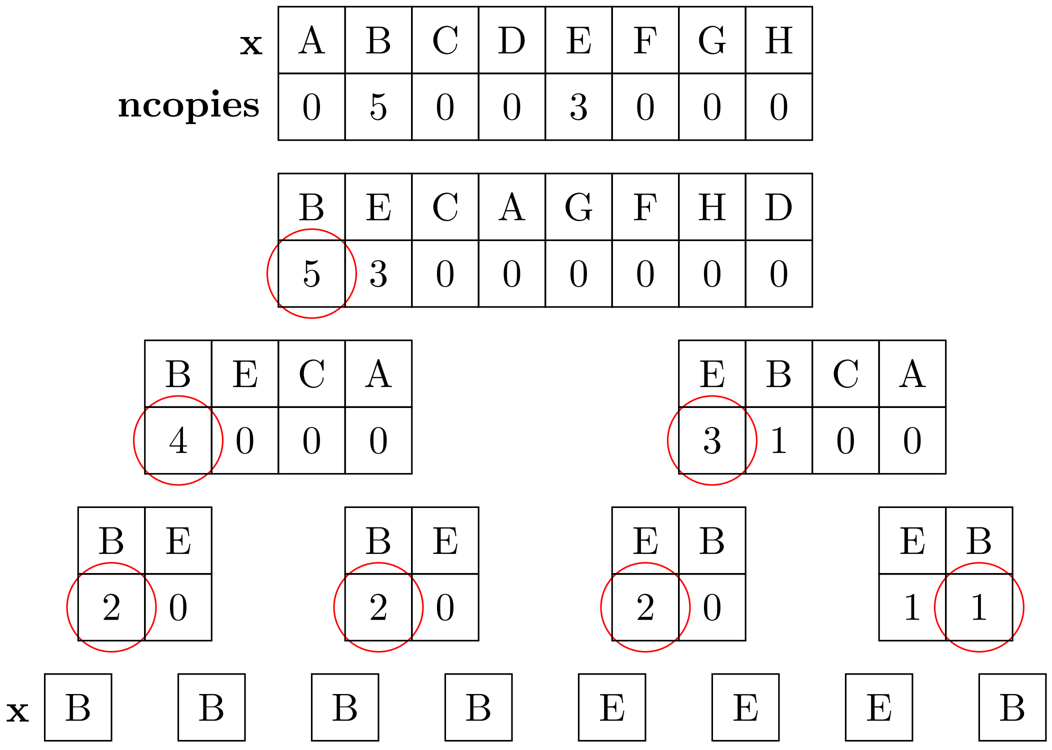


FIGURE 3.11: $O((\log_2 N)^3)$ Redistribute - example for $N = 8$ and $P = N$

3.3.4.2 Pivot Calculation

The pivot is the first index such that **csum** $\in \mathbb{Z}^N$, the Cumulative Sum over **ncopies**, is equal to $\frac{N}{2}$ or higher. Therefore, any given index i is *pivot* if the following logical expression is true:

$$\mathbf{csum}^i \geq \frac{N}{2} \land \mathbf{csum}^i - \mathbf{ncopies}^i < \frac{N}{2} \tag{3.4}$$

During the splitting phase, the cores send to its partner every i-th particle such that $\mathbf{ncopies}^i > 0$ and $i > pivot$. If $i = pivot$ and $\mathbf{csum}^i > \frac{N}{2}$, only the particle copies in excess must be sent: this translates to sending $\mathbf{csum}^i - \frac{N}{2}$ copies and keeping the remaining $\mathbf{ncopies}^i + \frac{N}{2} - \mathbf{csum}^i$ ones.

3.3.4.3 $O((\log_2 N)^3)$ Fully-Balanced Redistribute

Algorithm 17 describes an MPI pseudo-code in recursive form for the parallel redistribution described in this section. Since Bitonic Sort is performed $(\log_2 P)^2$ times we can infer that the achieved time complexity is $O((\log_2 N)^3)$, as previously mentioned in Section 3.2. A possible example for $N = 8$ and $P = N$ is illustrated in Figure 3.11, where the pivots are circled in red. The results for this redistribute implementation are provided in the following section, in comparison with other parallel redistribute implementation.

3.4 Bitonic Sort Based Redistribute on MPI

As anticipated in Section 3.2, the redistribute in [85] is presented on MapReduce and, although it has proven to be better than Algorithm 17, it has shown little to no speed-up vs S-R. This section shows how to implement on MPI the parallel redistribution in [85]. Then it repeats the same experiment in [85], with a view to showing that its disappointing results were mostly caused by the chosen framework.

Algorithm 17 uses Bitonic Sort to make sure the workload can be deterministically divided in less than $O(N)$. This is because, by sorting the particles we are also moving those ones for which $\mathbf{ncopies}^i > 0$ to one side of the node, leaving the other particles such that $\mathbf{ncopies}^i = 0$ to the other side of the node. Therefore, each i-th particle copy for $i \geq pivot$ can be safely sent to the other side of the node without colliding with other copies. However, Bitonic Sort is also the only reason why Algorithm 17 takes $O((\log_2 N)^3)$ computations.

To improve the time complexity, the parallel redistribution in [85] performs Bitonic Sort only once at the beginning. This guarantees that the particles that must be duplicated are separated from those that must be deleted. Then, the algorithm descends the same binary-tree structure as in Algorithm 17. At this point, the clever observation in [85] is that Bitonic Sort can be replaced by rotational shifts to ensure the workload distribution is deterministic. More precisely, each i-th particle such that $\mathbf{ncopies}^i > 0$ and $i \geq pivot$ is rotated by $r = (\frac{N}{2} - 1) - pivot$ positions. When operating on the binary tree, parallel Cumulative Sum is again computed at every stage to calculate the position of r. In this algorithm, it is necessary that all cores being aware of the exact position of the pivot, such that they can calculate the number of rotations r to be performed. Since the pivot could be located anywhere, whichever core finds *pivot* is tasked with

Algorithm 17 $O((\log_2 N)^3)$ MPI Redistribute

```
Input: $\mathbf{x}$, $\mathbf{ncopies}$, $N$, $P$, $n$, $p$, $comm$
Output: $\mathbf{x}$
1: if $N == n$ then
2:     $\mathbf{x} \leftarrow \texttt{S-R}(\mathbf{x}, \mathbf{ncopies}, n)$
3:     return $\mathbf{x}$
4: end if
5: Allocate $s$, an MPI_Status variable
6: $\texttt{MPI\_Bitonic\_Sort}(\mathbf{ncopies}, \mathbf{x}, N, P, p, comm)$
7: $\mathbf{csum} \leftarrow \texttt{MPI\_Cumulative\_Sum}(\mathbf{ncopies}, N, P, \texttt{MPI\_INT}, \texttt{Inclusive}, comm)$
8: $partner \leftarrow (p + \frac{P}{2})\&(P-1)$
9: for $i \leftarrow 0$; $i < N$; $i \leftarrow i+1$ do
10:     if $\mathbf{csum}^i < \frac{N}{2}$ then, $i < pivot$
11:         $\mathbf{copies\_to\_send}^i \leftarrow 0$
12:     else if $\mathbf{csum}^i \geq \frac{N}{2} \wedge \mathbf{csum}^i - \mathbf{ncopies}^i < \frac{N}{2}$ then, $i = pivot$
13:         $\mathbf{copies\_to\_send}^i \leftarrow \mathbf{csum}^i - \mathbf{ncopies}^i$
14:     else, $i > pivot$
15:         $\mathbf{copies\_to\_send}^i \leftarrow \mathbf{ncopies}^i$
16:     end if
17:     $\mathbf{ncopies}^i \leftarrow \mathbf{ncopies}^i - \mathbf{copies\_to\_send}^i$
18: end for
19: if $p < \frac{P}{2}$ then, I am a sender
20:     $\texttt{MPI\_Send}(\mathbf{copies\_to\_send}, n, \texttt{MPI\_INT}, partner, 0, comm)$
21:     $\texttt{MPI\_Send}(\mathbf{x}, n, \texttt{MPI\_DOUBLE}, partner, 0, comm)$
22: else, I am a receiver
23:     $\texttt{MPI\_Recv}(\mathbf{ncopies}, n, \texttt{MPI\_INT}, partner, 0, comm, s)$
24:     $\texttt{MPI\_Recv}(\mathbf{x}, n, \texttt{MPI\_DOUBLE}, partner, 0, comm, s)$
25: end if
26: $P \leftarrow \frac{P}{2}$, $N \leftarrow \frac{N}{2}$, $colour \leftarrow (int)(\frac{p}{P})$
27: $\texttt{MPI\_Comm\_split}(comm, colour, p, \&comm)$, split the communicator in two
28: $\texttt{MPI\_Comm\_size}(comm, \&P)$, register the new size of the communicator
29: $\texttt{MPI\_Comm\_rank}(comm, \&p)$, assign a new rank to each core
30: $O((\log_2 N)^3)$ $\texttt{MPI Redistribute}(\mathbf{x}, \mathbf{ncopies}, N, P, n, p, comm)$
```

broadcasting it to the other cores of the same node. The next two sections demonstrate that it is possible to broadcast the pivot and rotate the particles by r positions in $O(\log_2 N)$. Therefore, since Bitonic Sort is performed once and Cumulative Sum, pivot broadcast and rotational shifts are performed $\log_2 P$ times, we can infer that the parallel redistribution in [85] achieves $O((\log_2 N)^2)$ time complexity.

3.4.1 Pivot Broadcast

Once again, *pivot* is the first index such that $\mathbf{csum}^{pivot} \geq \frac{N}{2}$. Therefore, it can be found by checking the logical expression (3.4). However, as said in the previous section, *pivot* could be located anywhere between $i = 0$ and $i = \frac{N}{2} - 1$. This means any core with rank $p < \frac{P}{2}$ could find it. Therefore, it is necessary to make sure that *pivot* is available to all cores. This can be done by using a binary tree, akin to the one for reduction. During this operation, all but the owner of the pivot yield 0 towards the reduction while the owner yields the true value of $pivot$. This way, the output of reduction is guaranteed to be equal to the true pivot. Since (3.4) is embarrassingly parallel and the binary tree in reduction takes $\log_2 P$ partial Sums (see Section 3.3.2), we can infer that this routine achieves $O(\frac{N}{P} + \log_2 P)$ time complexity.

3.4.2 Rotational Shifts

In order to safely rotate the $\frac{N}{2}$ particles on the right side of $pivot$, the first thing to do is to save every i-th particle such that $i < pivot$ and as many copies as $\mathbf{ncopies}^i - \mathbf{csum}^i + \frac{N}{2}$ if $i = pivot$. This is to restore the saved particles once the rotations have been performed.

The cores within the same father node must now rotate the particles by $r = (\frac{N}{2} - 1) - pivot$ position, such that the father node is split into two balanced child nodes. Since $r \in \mathbb{Z}$, it can be expressed in base-two notation by using $\log_2 N$ bits. Therefore, one can decompose r into a sum of $\log_2 N$ power-of-two numbers (i.e. one per each bit of r) as follows:

$$r = \sum_{k=0}^{\log_2 N - 1} \mathbf{b}^k 2^k \tag{3.5}$$

where $\mathbf{b}^k$ is the k-th bit of r in base-two notation. One can then develop a bottom-up binary-tree structure, where at each stage a new bit of r is checked, starting from the Least Significant Bit (LSB) to the Most Significant Bit (MSB); the particles are then rotated by 2^k positions depending on whether $\mathbf{b}^k$ is 1 or 0. This way, the father node gets split in $O(\log_2 N)$ if $P = N$. In the more realistic case where $P < N$, an extra leaf stage must be performed before the cores starts ascending the binary tree. In this initial stage, internal rotations are performed locally in order to take care at once of the rotations related to the first $\log_2 \frac{N}{P}$ LSBs.

Figure 3.12 depicts a possible example for $N = 16$ particles and $P = 4$ cores rotating the particles by $r = 5$ positions. The scanned bits at each stage of the binary tree are in red.

3.4.3 Bitonic Sort Based Redistribute

Bitonic Sort is performed at the start to separate all the particles to be copied from those that must be deleted. Then, the particles can be split recursively in $O((\log_2 N)^2)$ by using sequences of Cumulative Sum over **ncopies**, pivot calculation and broadcast and rotational shifts in each node of a binary tree. After that, (3.2) is satisfied and S-R can be performed locally in $O(\frac{N}{P})$. As previously mentioned, the time complexity is $O((\log_2 N)^2)$ for $P = N$, but equal to (3.3) for any $P \leq N$ as Bitonic Sort is performed once. From now on, this algorithm will be referred to as Bitonic Sort Based Redistribute (B-R). A possible example for $N = 16$ and $P = 4$ is found in Figure 3.13, where the red circles represent node pivots and the arrows are applications of rotational shifts.

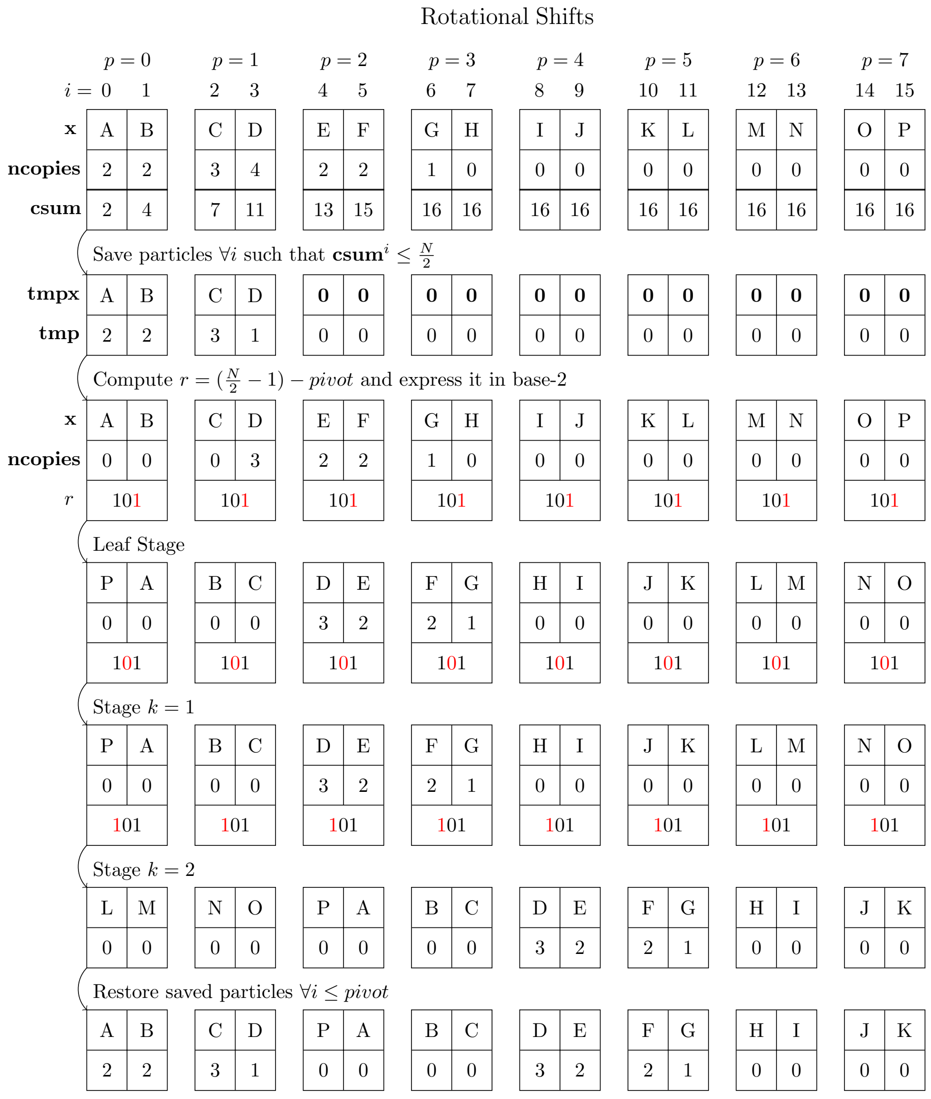

Rotational Shifts
p = 0 p = 1 p = 2 p = 3 p = 4 p = 5 p = 6 p = 7
i = 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
x A B C D E F G H I J K L M N O P
ncopies 2 2 3 4 2 2 1 0 0 0 0 0 0 0 0 0
csum 2 4 7 11 13 15 16 16 16 16 16 16 16 16 16 16
Save particles ∀i such that csum^i ≤ N/2
tmpx A B C D 0 0 0 0 0 0 0 0 0 0 0 0
tmp 2 2 3 1 0 0 0 0 0 0 0 0 0 0 0 0
Compute r = (N/2 − 1) − pivot and express it in base-2
x A B C D E F G H I J K L M N O P
ncopies 0 0 0 3 2 2 1 0 0 0 0 0 0 0 0 0
r 101 101 101 101 101 101 101 101
Leaf Stage
P A B C D E F G H I J K L M N O
0 0 0 0 3 2 2 1 0 0 0 0 0 0 0 0
101 101 101 101 101 101 101 101
Stage k = 1
P A B C D E F G H I J K L M N O
0 0 0 0 3 2 2 1 0 0 0 0 0 0 0 0
101 101 101 101 101 101 101 101
Stage k = 2
L M N O P A B C D E F G H I J K
0 0 0 0 0 0 0 0 3 2 2 1 0 0 0 0
Restore saved particles ∀i ≤ pivot
A B C D P A B C D E F G H I J K
2 2 3 1 0 0 0 0 3 2 2 1 0 0 0 0

FIGURE 3.12: Rotational Shifts - example for $N = 16$, $P = 4$ and $r = 5$

3.4.4 Algorithmic Implementation

This section provides some algorithmic implementation details about B-R and its key components discussed in Sections 3.4.1 and 3.4.2.

Algorithm 18 summarises the necessary steps to implement `Pivot_Bcast`. Each core searches for *pivot* using an embarrassingly parallel for loop. The core p that finds it also needs to convert it to a global index by rescaling it by $p \times n$, i.e. the total number of memory locations belonging to the cores with lower ranks. At this point any core could have found *pivot*. Therefore, standard all-to-one MPI broadcast routines, such as `MPI_Bcast`, cannot be used to to broadcast the position of *pivot*. Here, `MPI_Allreduce` is used instead.

Algorithm 19 illustrates an MPI pseudo-code performing r rotational shifts. As anticipated in Section 3.4.2, this routine needs two embarrassingly parallel for loop, one at the beginning and one at the end, to save and restore the particles that must not

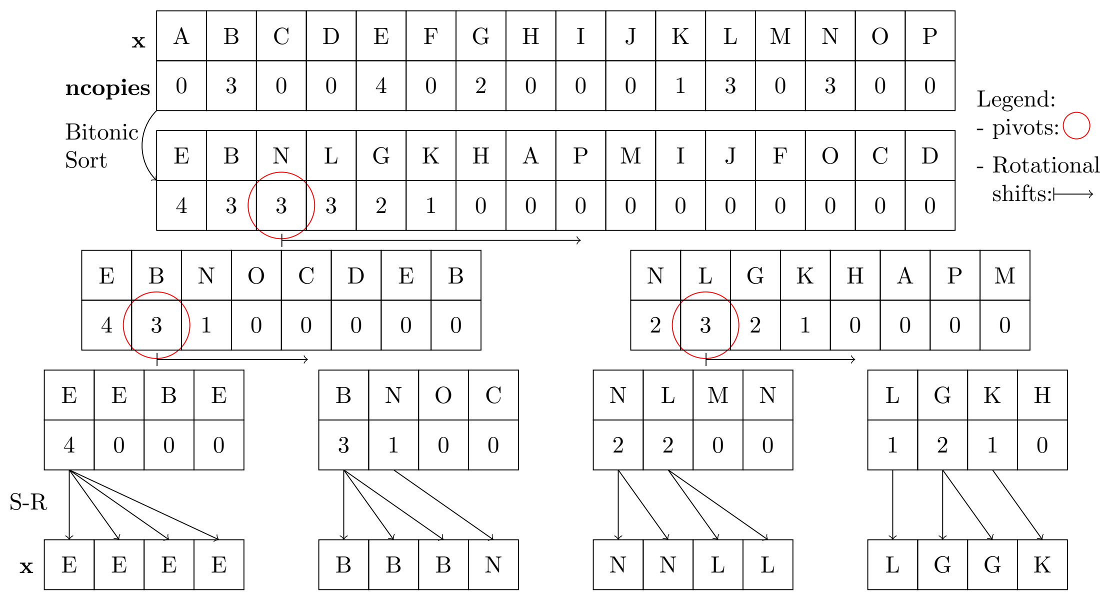


FIGURE 3.13: Bitonic Sort Based Redistribute - example for $N = 16$ and $P = 4$

Algorithm 18 Pivot Bcast

Input: **ncopies**, **csum**, N, P, *comm*

Output: *pivot*

```
1: $n \leftarrow \frac{N}{P}$
2: $idx \leftarrow 0$
3: for $i \leftarrow 0$; $i < n$; $i \leftarrow i + 1$ do
4:     if $\mathbf{csum}^i \geq \frac{N}{2} \wedge \mathbf{csum}^i - \mathbf{ncopies}^i < \frac{N}{2}$ then
5:         $idx \leftarrow p \times n + i$, the pivot must be expressed as a global index
6:         break
7:     end if
8: end for
9: MPI_Allreduce($idx, pivot, 1,$ MPI_INT, MPI_SUM, $comm$)
```

move. In between, the cores are organised in a binary tree for loop where they exchange particles by using `MPI_Sendrecv` at each stage.

Algorithm 20 describes an MPI pseudo-code for the parallel redistribution in [85]. As we can see, this routine performs Bitonic Sort once, followed by a binary-tree step performing Algorithms 14, 18 and 19 in sequence at each stage. The routine is then finalised by using S-R to redistribute the particles within the memory of each core.

The next section repeats the experiment in [85] which compares B-R with Algorithm 17 and the Centralised Redistribute (C-R), i.e. the redistribute parallelisation in Centralised Resampling, whose pseudo-code can be found in Algorithm 21.

3.4.5 Numerical Results

This section first compares single iterations of the redistributions in Algorithms 17, 20 and 21. Then, it studies the impact of each of these variants on the overall run-time of a SIR PF working on an exemplary dynamic model.

Algorithm 19 MPI Rotational Shifts

```
Input:  $\mathbf{x}$, $\mathbf{ncopies}$, $\mathbf{csum}$, $N$, $P$, $pivot$, $comm$
Output:  $\mathbf{x}$, $\mathbf{ncopies}$
1: $n \leftarrow \frac{N}{P}$
2: $r \leftarrow pivot - (\frac{N}{2} - 1)$
3: for $i \leftarrow 0$; $i < n$; $i \leftarrow i + 1$ do, save particles on the right side of $pivot$
4:     if $p \times n + i < pivot$ then
5:         $\mathbf{tmp}^i \leftarrow \mathbf{ncopies}^i$
6:         $\mathbf{tmpx}^i \leftarrow \mathbf{x}^i$
7:     else if $p \times n + i > pivot$ then
8:         $\mathbf{tmp}^i \leftarrow 0$
9:         $\mathbf{tmpx}^i \leftarrow \mathbf{0}$
10:    else
11:        $\mathbf{tmp}^i \leftarrow \mathbf{csum}^i - \frac{N}{2}$
12:        $\mathbf{ncopies}^i \leftarrow \mathbf{ncopies}^i - \mathbf{tmp}^i$
13:        $\mathbf{tmpx}^i \leftarrow \mathbf{x}^i$
14:    end if
15: end for
16: if $P < N$ then Leaf stage
17:    if $r\&(n-1)$ then The $k$-th bit is 1
18:        $sp \leftarrow (p+1)\&(P-1)$, compute the sending partner
19:        $rp \leftarrow (p-1)\&(P-1)$, compute the receiving partner
20:        for $i \leftarrow 0$; $i < n$; $i \leftarrow i + 1$ do
21:            $j \leftarrow (r + r\&(n-1))\&(n-1)$
22:            $\mathbf{tmp2}^j \leftarrow \mathbf{ncopies}^i$
23:            $\mathbf{tmpx2}^j \leftarrow \mathbf{x}^i$
24:        end for
25:        $\mathbf{ncopies}^{0:n-1} \leftarrow \mathbf{tmp}^{0:n-1}$
26:        $\mathbf{x}^{0:n-1} \leftarrow \mathbf{tmpx2}^{0:n-1}$
27:        $\mathtt{MPI\_Sendrecv}(\mathbf{ncopies}, n, \mathtt{MPI\_INT}, sp, 0, \mathbf{tmp2}, n, \mathtt{MPI\_INT}, rp, 0, comm, s)$
28:        $\mathtt{MPI\_Sendrecv}(\mathbf{x}, n, \mathtt{MPI\_DOUBLE}, sp, 0, \mathbf{tmpx2}, n, \mathtt{MPI\_DOUBLE}, rp, 0, comm, s)$
29:    end if
30: end if
31: for $k \leftarrow \log_2 n$; $k < \log_2 r$; $k \leftarrow k + 1$ do Binary tree
32:    if $r\&2^k$ then The $k$-th bit is 1
33:        $sp \leftarrow (p + \frac{P}{2^k})\&(P-1)$, compute the sending partner
34:        $rp \leftarrow (p - \frac{P}{2^k})\&(P-1)$, compute the receiving partner
35:        $\mathtt{MPI\_Sendrecv}(\mathbf{ncopies}, n, \mathtt{MPI\_INT}, sp, 0, \mathbf{tmp2}, n, \mathtt{MPI\_INT}, rp, 0, comm, s)$[3]
36:        $\mathtt{MPI\_Sendrecv}(\mathbf{x}, n, \mathtt{MPI\_DOUBLE}, sp, 0, \mathbf{tmpx2}, n, \mathtt{MPI\_DOUBLE}, rp, 0, comm, s)$
37:    end if
38: end for
39: for $i \leftarrow 0$; $i < n$; $i \leftarrow i + 1$ do, restore particles on the right side of $pivot$
40:    $\mathbf{ncopies}^i \leftarrow \mathbf{ncopies}^i + \mathbf{tmp}^i$, $\mathbf{x}^i \leftarrow \mathbf{x}^i + \mathbf{tmpx}^i$
41: end for
```

3.4.5.1 Redistribute

Figure 3.14 shows the run-times for all three redistribute algorithms that have been described in Sections 3.3.4 and 3.4.3. To ensure fair comparison, all algorithms have

Algorithm 20 Bitonic sort based Redistribute (B-R)

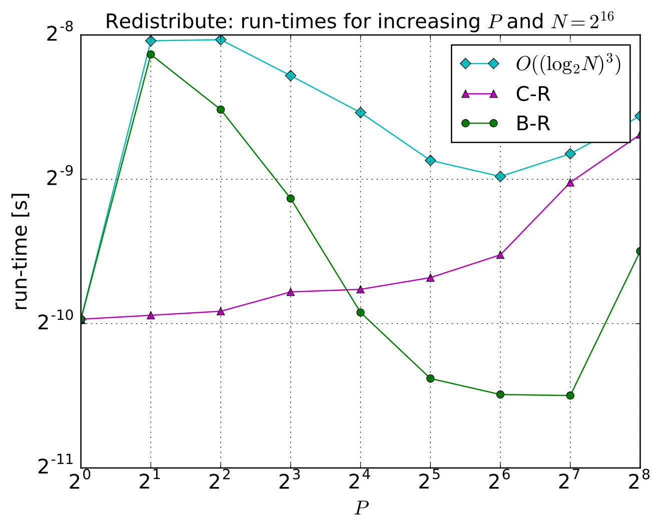
```
Input: $\mathbf{x}$, $\mathbf{ncopies}$, $N$, $P$, $n = \frac{N}{P}$, $comm$
Output: $\mathbf{x}$
1: if $P > 1$ then, sort the particles
2:     MPI_Bitonic_Sort($\mathbf{x}, \mathbf{ncopies}, N, P, comm$)
3: end if
4: for $k \leftarrow 1$; $k \leq \log_2 P$; $k \leftarrow k+1$ do, Binary tree
5:     $\mathbf{csum} \leftarrow$ MPI_Cumulative_Sum($\mathbf{ncopies}, N, P, comm$)
6:     $pivot \leftarrow$ Pivot_Bcast($\mathbf{ncopies}, \mathbf{csum}, N, P, comm$)
7:     $\mathbf{x}, \mathbf{ncopies} \leftarrow$MPI_Rot_Shifts($\mathbf{x}, \mathbf{ncopies}, \mathbf{csum}, N, P, pivot, comm$),
8:     $N \leftarrow N/2$
9:     $P \leftarrow P/2$
10:    $colour \leftarrow (int)(\frac{p}{P})$
11:    MPI_Comm_split($comm, colour, p, \&comm$), split the communicator in two
12:    MPI_Comm_size($comm, \&P$), register the new size of the communicator
13:    MPI_Comm_rank($comm, \&p$), assign a new rank to each core
14: end for, $\mathbf{ncopies}$ now complies with (3.2)
15: $\mathbf{x} \leftarrow$ S-R($\mathbf{x}, \mathbf{ncopies}, n$)
```

Algorithm 21 Centralised Redistribute (C-R)

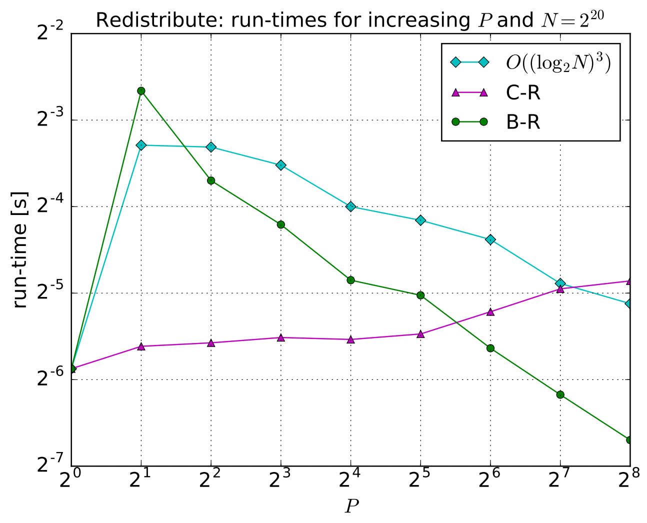
```
Input: $\mathbf{x}$, $\mathbf{ncopies}$, $N$, $P$, $p$
Output: $\mathbf{x}$
1: $n \leftarrow \frac{N}{P}$
2: MPI_Gather($\mathbf{ncopies}, n$, MPI_INT, $\mathbf{ncopies}, n$, MPI_INT, 0, MPI_COMM_WORLD)
3: MPI_Gather($\mathbf{x}, n$, MPI_DOUBLE, $\mathbf{x}, n$, MPI_DOUBLE, 0, MPI_COMM_WORLD)
4: if $p == 0$ then
5:     $\mathbf{x} \leftarrow$ S-R($\mathbf{x}, \mathbf{ncopies}, N$)
6: end if
7: MPI_Scatter($\mathbf{x}, n$, MPI_DOUBLE, $\mathbf{x}, n$, MPI_DOUBLE, 0, MPI_COMM_WORLD)
```

always been tested for the same input pair: **x**, **ncopies**. To guarantee (2.17), **ncopies** is generated randomly by using MVR with a normally distributed input $\tilde{\mathbf{w}}$, as this is a common case for several exemplary models. As can be observed, C-R is very fast for a few cores, but becomes progressively slower as P increases, primarily due to increasing cost of communication. The run-times for the $O((\log_2 N)^2)$ and $O((\log_2 N)^3)$ variants increase rapidly for $P = 2$ MPI cores, because neither Bitonic Sort nor the binary-tree phase are needed when $P = 1$. However, with the number of MPI processes increasing, it is evident that B-R outperforms Algorithm 17 for any $P > 2$, and eventually outperforms C-R as well starting from at least $P = 64$ MPI cores. Also, B-R provides up to a four time speed-up vs the other two redistribute variants.

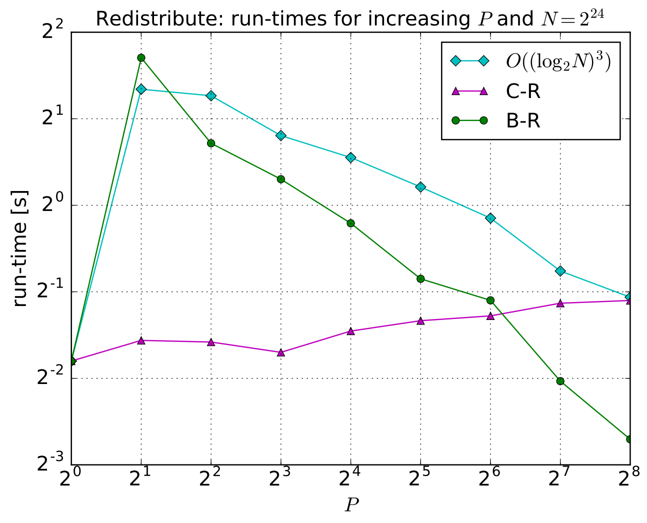
With these results in view, we can expect to see that the new algorithm will lead to slow-downs for any $P < 64$ MPI processes when applied to the context of SMC methods. These are not further discussed here. Instead more relevant results about the overall speed-up for the PF will be discussed in the section that follows this.



(A) $N = 2^{16}$



(B) $N = 2^{20}$



(C) $N = 2^{24}$

FIGURE 3.14: B-R vs C-R vs $O((\log_2 N)^3)$ Redistribute - run-times for increasing N and P

3.4.5.2 Stochastic Volatility

The chosen example for this experiment is a stochastic volatility model which appeared several times in the literature [29, 30] and estimates the pound-to-dollar exchange rate between October 1, 1981 and June 28, 1985. In [29], this model has been used to demonstrate the utility of advanced SMC methods, such as Block Sampling PFs, over SIR PFs. The dynamic model follows:

$$\mathbf{X}_t = \phi \mathbf{X}_{t-1} + \sigma \mathbf{V}_t \tag{3.6a}$$

$$\mathbf{Y}_t = \beta \exp\left(\frac{\mathbf{X}_t}{2}\right) \mathbf{W}_t \tag{3.6b}$$

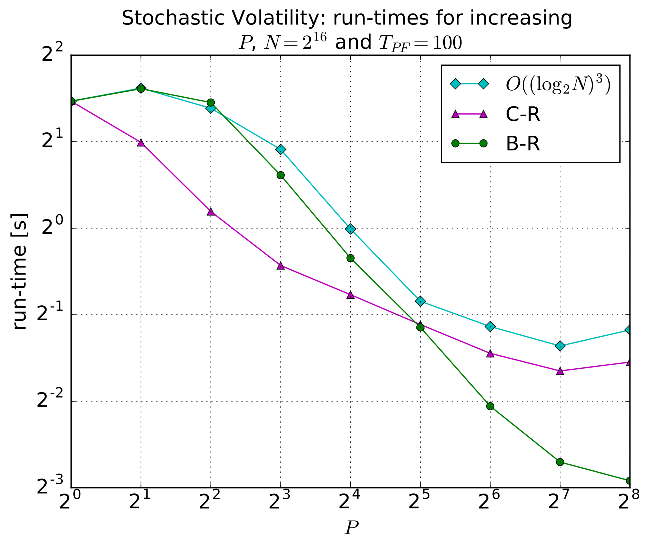
where the coefficients $\phi = 0.9731$, $\sigma = 0.1726$, $\beta = 0.6338$ (as selected in [29]) and $\mathbf{V}_t \sim \mathcal{N}(0,1)$ and $\mathbf{W}_t \sim \mathcal{N}(0,1)$. The initial state is sampled as $\mathbf{X}_0 \sim \mathcal{N}(0, \frac{\sigma^2}{1-\phi^2})$. The particles are initially drawn from $p(\mathbf{X}_0)$ and then from dynamic model. Hence, (2.13) is simplified to $\mathbf{w}_t^i = \mathbf{w}_{t-1}^i p\left(\mathbf{Y}_t | \mathbf{x}_t^i\right)$.

For this experiment, three different versions of Algorithm 1 are used, only differing for the type of redistribute parallelisation in use. Each run-time has been taken for 100 consecutive time steps and, in order to compare the algorithms accurately, resampling is purposely computed every time, to ensure the frequency of redistribution is the same.

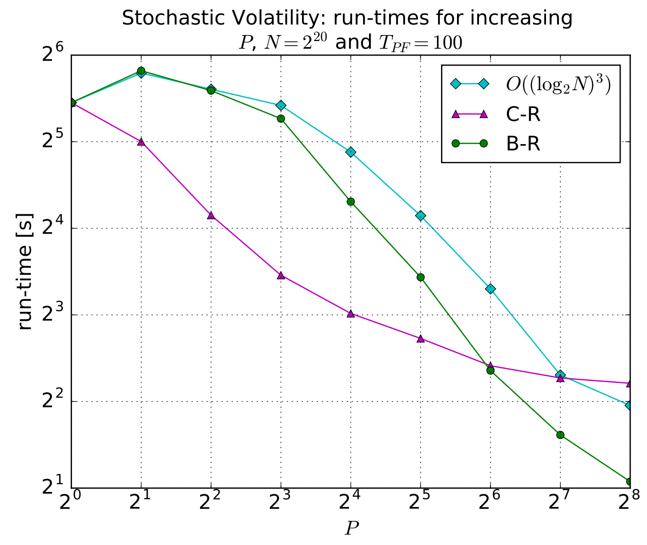
Figure 3.15 shows that the speed of PF using C-R improves for a limited number of cores. This is because C-R is faster than other tasks, such as `MVR`, when P is low. However, when P is high enough all tasks become faster than redistribute. At this point, C-R emerges as the bottleneck and then the PF stops scaling. On the other hand, the PF using B-R and the $O((\log_2 N)^3)$ PF scale progressively for $P > 2$ cores. However, the most interesting result is that not only is the $O((\log_2 N)^2)$ PF faster than the $O((\log_2 N)^3)$ PF for any DOP (as expected from [85]) but, most importantly, it also outperforms the $O(N)$ PF for $P = 64, 128$, depending on N. These results prove that MPI is a better environment than MapReduce for the PF with B-R. More precisely, on MPI the $O((\log_2 N)^2)$ PF provides about a two-fold speed-up vs the centralised PF for $P = 256$ cores, while on MapReduce it was much slower for $P = 512$ cores.

The same figure underlines that the PF with B-R for $P = 256$ is up to 30 times faster than each implementation of the PF running on a single core, as the three PF algorithms are equivalent when $P = 1$.

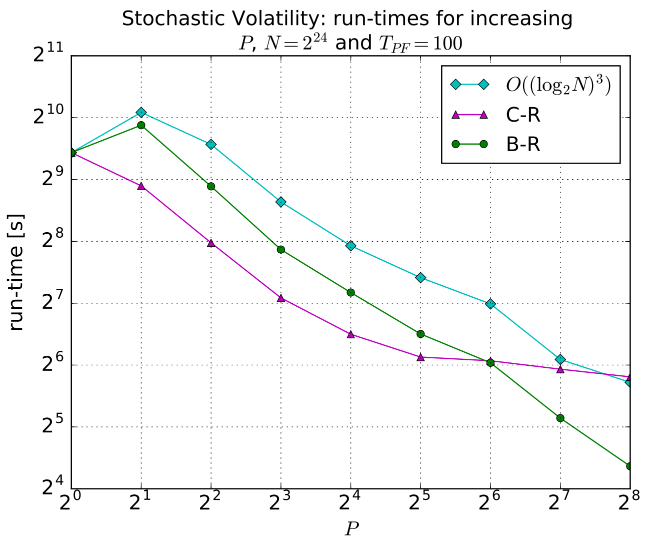
These findings are encouraging and motivates further investigation of novel MPI solutions to optimise the results above. Although B-R is indeed the best fully-balanced redistribute which has been described so far, as we can see in Figure 3.16, it still is a significant bottleneck for any $P > 1$, when it comes to its application in the context of SIR PFs. The next section presents a novel variant of B-R that is effective for low DOP, while Chapter 4 focuses on a novel redistribute which is fast for any P.



(A) $N = 2^{16}$



(B) $N = 2^{20}$



(C) $N = 2^{24}$

FIGURE 3.15: Stochastic Volatility - run-times of PF with B-R or C-R or $O((\log_2 N)^3)$ redistribute for increasing N and P

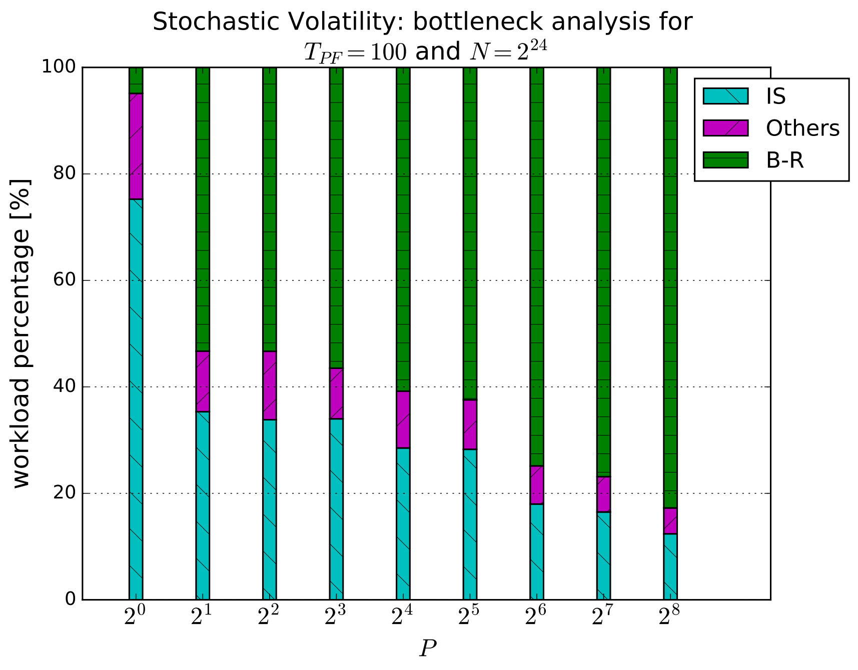


FIGURE 3.16: Stochastic Volatility - bottleneck analysis for PF with B-R for $N = 2^{24}$

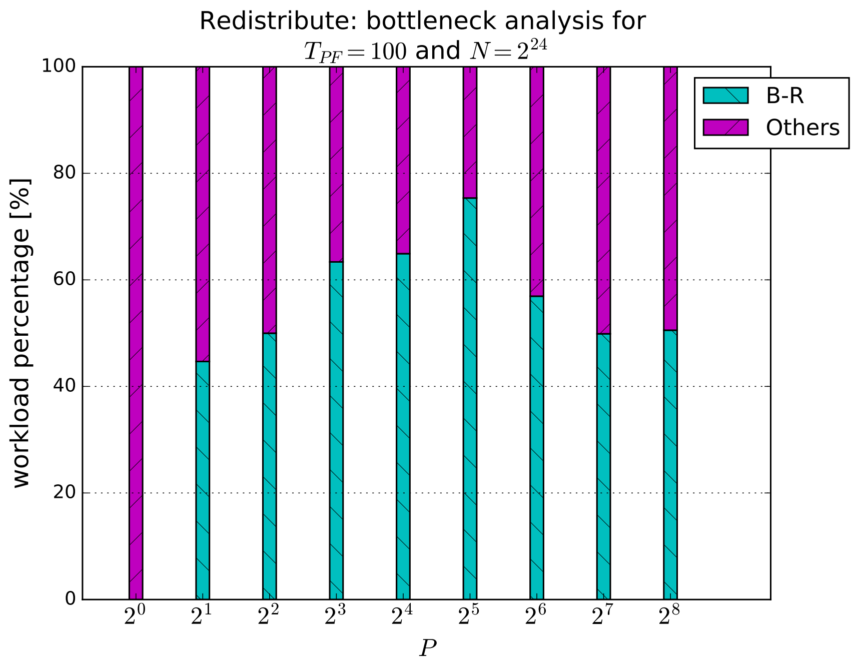


FIGURE 3.17: B-R - bottleneck analysis for $N = 2^{24}$

3.5 Nearly Sort Based Redistribute on MPI

The previous section has proven that B-R is the bottleneck of PF. In order to effectively optimise the overall performance, a full profiling of Algorithm 20 is required to identify which among Bitonic Sort, S-R or the other tasks in B-R is on the most computationally intensive. As can be observe in Figure 3.17, Bitonic Sort always accounts for at least 50% of total run-time of B-R for $P > 1$. Therefore, the sorting phase needs revision both for low and high DOP. The rest of this chapter focuses on improvements for low DOP, while the next chapter is focused on providing improvements for any P.

3.5.1 Alternative Single Core Sorting Algorithms

One possible way to improve Bitonic Sort (and by extension redistribute) is to substitute the serial Bitonic Sort algorithm with a better single-core sorting algorithm. In the literature, there are plenty of alternatives to Bitonic Sort available. Algorithms such as Quicksort [42], Mergesort [4] and Heapsort [80], for example, achieve $O(N \log_2 N)$ time complexity. Quicksort is on average faster than Mergesort and Heapsort. However, Quicksort's choice of its pivot can severely influence the performance: it is known, in fact, that Quicksort's worst-case time complexity is $O(N^2)$. This occurs when the pivot chosen at every iteration is equal to either the minimum or the maximum of the available keys. Although this case is statistically very rare in several modern applications, in the case of SMC methods the worst-case scenario is however often encountered: **ncopies** has to be sorted and, since (2.17) holds, there is a high probability that 0 is picked as Quicksort's pivot, i.e. a high probability that the pivot is the minimum element.

Heapsort achieves $O(N \log_2 N)$ time complexity in all cases except when all keys are equal. In this special although rather unlikely case, the time complexity is $O(N)$. However, Mergesort is perfectly deterministic and data-independent and represents a valid alternative to Bitonic Sort to fit a fully-balanced redistribute. A Bitonic Sorter with Mergesort performed locally achieves the following time complexity:

$$O\left(\frac{N}{P}\log_2\left(\frac{N}{P}\right) + \frac{N}{P}(\log_2 P)^2\right) \tag{3.7}$$

We also observe that **ncopies** is an array of integers. Hence, one could use locally linear time sorting algorithms such as Counting Sort [32] or Radix Sort [5] (which are both only applicable to arrays of integers). Although Counting Sort has deterministic and data-independent time complexity, its space complexity is data-dependent. This is because Counting Sort allocates a temporary array with as many elements as $max - min + 1$. In the worst-case $max = N$, $min = 0$ and since N in extreme scenarios could be very high, the temporary array may not fit within the local memory of a single machine. This problem is shared with C-R. On the other hand, Radix Sort is a feasible deterministic solution. However, Radix Sort is data-dependent because its time complexity is actually $O(C \cdot N)$ where the constant C is equal to the number of digits of the maximum element (which can be N in the worst-case). Therefore, Radix Sort may be too slow when N is high and its run-time may fluctuate too much as a function of the input.

In summary, since this thesis is mostly interested in fully-balanced solutions, it is necessary to find a parallel sorting algorithm that works with integer numbers, and is deterministic and data-independent with respect to both time and space complexity. While a combination of Bitonic Sort and Mergesort within each core achieves these aims, the next two sections develop an improved strategy that is sufficient for the needs of this thesis and does not require sort at all.

3.5.2 Sequential Nearly Sort

The replacement of sort with rotational shift in B-R has improved the time complexity from $O((\log_2 N)^3)$ to $O((\log_2 N)^2)$. However, it has also led to a more subtle consideration: by observing the input of rotational shifts we can infer that one does not actually need to perfectly sort the particles to divide the workload deterministically. This condition is always satisfied as long as stage by stage the particles that have to be duplicated are separated from those that do not. To make things more clear it is necessary to first provide the following definition.

Definition 3.2. A sequence of N non-negative integer **ncopies** is nearly-sorted in descending order when it has the following shape:

$$\mathbf{ncopies} = \left[\lambda^0, \lambda^1, ..., \lambda^{m-1}, 0, \ldots, 0\right] \tag{3.8}$$

where $\mathbf{ncopies}^i > 0 \ \forall i = 0, 1, ..., m-1$ and $0 \leq m \leq N$. On the other hand, **ncopies** is an ascending nearly-sorted sequence if the last m elements are positive and the first are 0.

We can infer that the workload can be divided deterministically if **ncopies** is a nearly-sorted sequence. In B-R, this condition is ensured by sorting before the subsequent parts of the redistribute step, consisting of Algorithms 14, 18 and 19. While there are single-core sorting algorithms that achieve $O(N)$ time complexity, these algorithms do not meet the constraints of deterministic run-time and storage. However, it is possible to use a single core Nearly Sort for an array of integers with a deterministic and data-independent approach with $O(N)$ time complexity.

The key idea simply consists of checking the value of each $\mathbf{ncopies}^i$. If this value is positive, the core will copy $\mathbf{x}^i$ to the left of an output array; if $\mathbf{ncopies}^i = 0$, $\mathbf{x}^i$ will be copied to right. Each cores repeats this atomic operation for each particle it owns; after that, the output array is necessarily nearly-sorted in descending order. For ascending order outputs, the logic of the atomic operation must be reversed. This thesis refers to the described routine as Sequential Nearly Sort (S-NS). To complete S-NS, each core requires one atomic operation per particle that it owns, which means that S-NS achieves $O(N)$ time complexity or $O(\frac{N}{P})$ if we consider that each core owns $n = \frac{N}{P}$ elements.

S-NS is, therefore, a very good alternative to Serial Bitonic Sort, Mergesort, Heapsort and Radix Sort. This is because it achieves low time complexity with deterministic and data-independent run-time and space complexity. The next section shows how to use S-NS in a parallel algorithm that improves the performance of Bitonic Sort.

3.5.3 Parallel $O((\log_2 N)^2)$ Nearly Sort

The goal of this section is show how S-NS can be used as part of a parallel algorithm which generates a nearly-sorted sequence from a random input one. In this section and the following one, it is discussed how to achieve this.

Let $\mathbf{h}$ be a sequence of N elements. $\mathbf{h}$ is called a nearly-bitonic sequence when it is possible to find an index k which splits $\mathbf{h}$ into two monotonic nearly-sorted sequences. One could use S-NS and the same sorting network of Bitonic Sort to first divide the input into a series of nearly-bitonic sequences, and then to recursively merge the sequences together until we generate a monotonic nearly-sorted sequence at the last step.

To achieve that, `Merge` in Algorithm 16 requires adaptation such that it can process a nearly-bitonic sequence and returns a monotonic nearly-sorted sequence. This algorithm has been named Nearly Merge. Stage-by-stage, one core with MPI rank i is coupled with another core with MPI rank j. The assumption is that each core owns a nearly-sorted sequence of keys such that the combination of both is necessarily a nearly-bitonic sequence. Stage by stage, the cores exchange their local data. Then they consume a complementary subset of $\frac{N}{P}$ elements. Depending on the direction of the arrow in the sorting network (see again Figure 3.9) and its rank compared to its partner's rank, one core will start consuming the 0s first and then the positive elements, while the other core will do the opposite. This way, the 0s will be confined to one end of the output array, separated from the positive elements.

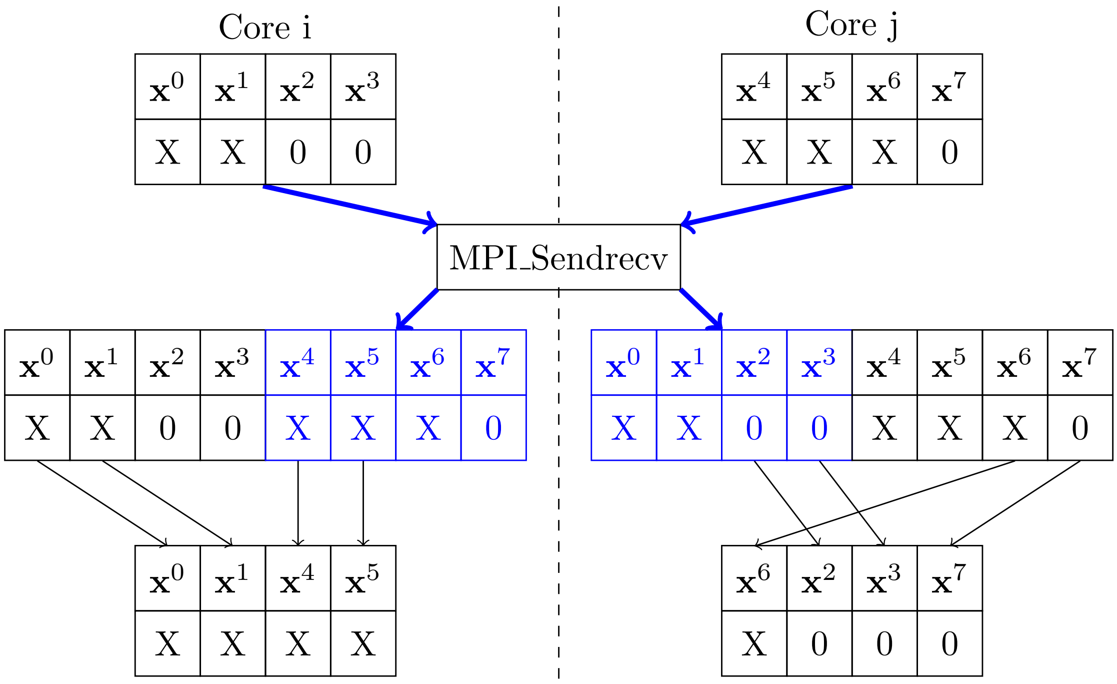

Core i
Core j
MPI_Sendrecv
$\mathbf{x}^0$ $\mathbf{x}^1$ $\mathbf{x}^2$ $\mathbf{x}^3$
X X 0 0
$\mathbf{x}^4$ $\mathbf{x}^5$ $\mathbf{x}^6$ $\mathbf{x}^7$
X X X 0
$\mathbf{x}^0$ $\mathbf{x}^1$ $\mathbf{x}^4$ $\mathbf{x}^5$
X X X X
$\mathbf{x}^6$ $\mathbf{x}^2$ $\mathbf{x}^3$ $\mathbf{x}^7$
X 0 0 0

FIGURE 3.18: Nearly Merge - example for $\frac{N}{P} = 4$ and $up = 1$

Figure 3.18 illustrates a possible example of Nearly Merge, where the two coupled cores own 4 keys; the positive elements are padded with Xs for brevity. By extension, each core owns exactly $\frac{N}{P}$ particles and performs the same amount of writes to memory.

Therefore, Nearly Merge achieves $O(\frac{N}{P})$ time complexity just as S-NS does. Considering these conclusions, we can infer that, by switching `Serial_Bitonic_Sort` and `Merge` in Algorithm 16 with S-NS and Nearly Merge, we obtain a parallel implementation of Nearly Sort which takes

$$O\left(\frac{N}{P} + \frac{N}{P}(\log_2 P)^2\right) \tag{3.9}$$

comparisons. A possible example is found in Figure 3.19.

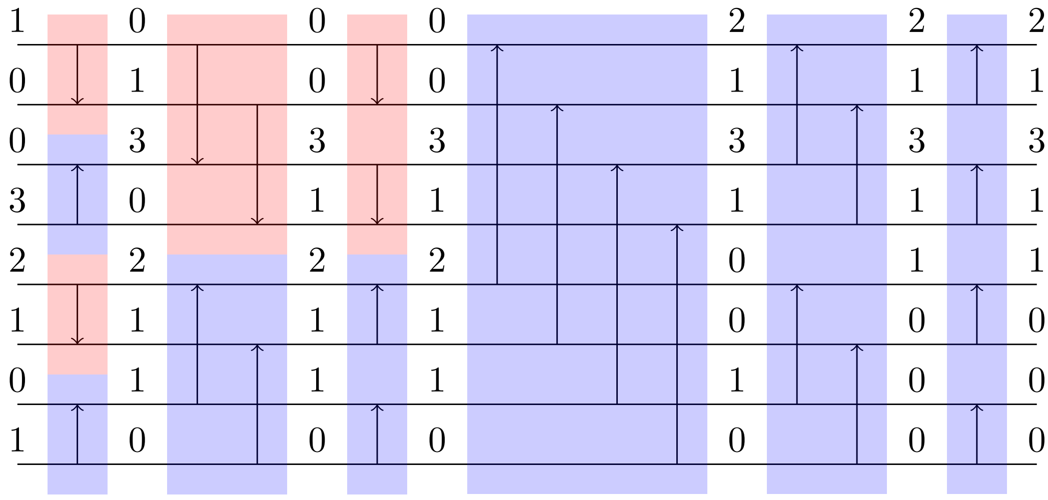


Figure 3.19: Nearly Sort - sorting network

3.5.4 Nearly Sort Based Redistribute

Nearly Sort has asymptotically the same time complexity of Bitonic Sort when $P = N$, but the time complexity for the serial algorithm is improved by a factor of $O((\log_2(\frac{N}{P}))^2)$. Therefore, we can expect this algorithm to outperform Bitonic Sort. By extension, if Bitonic Sort is replaced with Nearly Sort in B-R, we can also expect to have better performance. A possible example for $N = 16$ and $P = 4$ is shown in Figure 3.20. From now on, this algorithm is referred to as Nearly Sort Based Redistribute (N-R).

3.5.5 Algorithmic Implementation

This section provides some algorithmic implementation details for N-R and its key components on MPI.

Algorithm 22 illustrates S-NS[4], declares two iterators which respectively point at the first and the last element of **ncopies**. Step by step, the i-th element of **ncopies** is considered and if the value is positive then the particle is copied to the left end of the output array. If not, it gets copied to the right end. The output $\textbf{ncopies}_{new}$ will then be a descending nearly-sorted sequence.

[4]The pseudo-code for S-NS outputs **zeros**, the total count of $\textbf{ncopies}^i = 0$ elements. The reason for that will become clear in Chapter 4, although it seems useless at this point of the narrative.

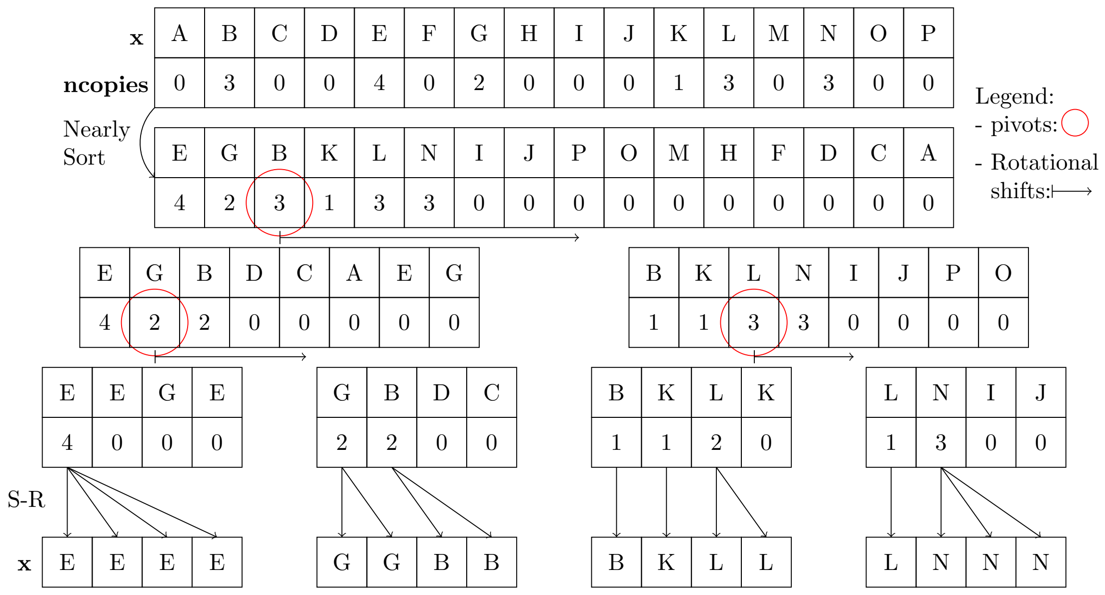


FIGURE 3.20: Nearly Sort Based Redistribute - example for $N = 16$ and $P = 4$

Algorithm 22 Sequential Nearly Sort (S-NS)

Input: **x**, **ncopies**, n

Output: $\mathbf{x}_{new}$, $\mathbf{ncopies}_{new}$, **zeros**

```
1:  $l \leftarrow 0$
2:  $r \leftarrow n - 1$
3:  for $i \leftarrow 0; i < n; i \leftarrow i + 1$ do
4:      if $\mathbf{ncopies}^i > 0$ then
5:          $\mathbf{ncopies}^l_{new} \leftarrow \mathbf{ncopies}^i$
6:          $\mathbf{x}^l_{new} \leftarrow \mathbf{x}^i$
7:          $l \leftarrow l + 1$
8:      else
9:          $\mathbf{ncopies}^r_{new} \leftarrow \mathbf{ncopies}^i$
10:         $\mathbf{x}^r_{new} \leftarrow \mathbf{x}^i$
11:         $r \leftarrow r - 1$
12:     end if
13: end for
14: zeros $\leftarrow n - l$
```

An MPI implementation of Nearly Sort is illustrate by Algorithm 23. As we can see, this algorithm have a similar body of instructions to 16. At each stage, the cores call `MPI_Sendrecv` to exchange data with their partners. Then each core will invoke a Nearly Merge routine to either consume the 0s first and then the positive elements (`Nearly_Merge_Down`) or the opposite (`Nearly_Merge_Up`), depending on the binary tree stage and the partner's rank.

Algorithm 24 describes N-R, which is essentially the same as 20 with `MPI_Bitonic_Sort` being replaced by `MPI_Nearly_Sort`.

Algorithm 23 MPI Nearly Sort

```
Input: x, ncopies, $N$, $P$, $p$, $comm$
Output: x, ncopies
1: $n \leftarrow \frac{N}{P}$
2: Allocate $s$, an MPI_Status variable
3: S-NS(x, ncopies, $n$)
4: for $i \leftarrow 2$; $i \leq P$; $i \leftarrow 2 \cdot i$ do
5:     $up \leftarrow$ Direction($p, i$)
6:     for $j \leftarrow 0$; $j < \log_2 i$; $j \leftarrow j + 1$ do
7:         $par \leftarrow$ PartnerCalc($p, i, j$)
8:         MPI_Sendrecv(ncopies, $n$, MPI_INT, $par$, 0, tmp, $n$, MPI_INT, $par$, 0, $comm$, $s$)
9:         MPI_Sendrecv(x, $n$, MPI_DOUBLE, $par$, 0, tmpx, $n$, MPI_DOUBLE, $par$, 0, $comm$, $s$)
10:        if $up$ then, The arrow in Figure 3.19 is pointing up
11:            if $p < par$ then, consume the positive keys first
12:                Nearly_Merge_Up(x, ncopies, tmp, tmpx, $n$)
13:            else, consume the 0s first
14:                Nearly_Merge_Down(x, ncopies, tmp, tmpx, $n$)
15:            end if
16:        else, The arrow in Figure 3.19 is pointing down
17:            if $p < par$ then
18:                Nearly_Merge_Down(x, ncopies, tmp, tmpx, $n$)
19:            else
20:                Nearly_Merge_Up(x, ncopies, tmp, tmpx, $n$)
21:            end if
22:        end if
23:    end for
24: end for
```

Algorithm 24 Nearly sort based Redistribute (N-R)

```
Input: x, ncopies, $N$, $P$, $n = \frac{N}{P}$, $comm$
Output: x
1: if $P > 1$ then, sort the particles
2:     MPI_Nearly_Sort(x, ncopies, $N$, $P$, $comm$)
3: end if
4: for $k \leftarrow 1$; $k \leq \log_2 P$; $k \leftarrow k + 1$ do, Binary tree
5:     csum $\leftarrow$ MPI_Cumulative_Sum(ncopies, $N$, $P$, $comm$)
6:     $pivot \leftarrow$ Pivot_Bcast(ncopies, csum, $N$, $P$, $comm$)
7:     x, ncopies $\leftarrow$MPI_Rot_Shifts(x, ncopies, csum, $N$, $P$, $pivot$, $comm$),
8:     $N \leftarrow N/2$
9:     $P \leftarrow P/2$
10:    $colour \leftarrow (int)(\frac{p}{P})$
11:    MPI_Comm_split($comm$, $colour$, $p$, &$comm$), split the communicator in two
12:    MPI_Comm_size($comm$, &$P$), register the new size of the communicator
13:    MPI_Comm_rank($comm$, &$p$), assign a new rank to each core
14: end for, ncopies now complies with (3.2)
15: x $\leftarrow$ S-R(x, ncopies, $n$)
```

3.5.6 Numerical Results

This section first compares Nearly Sort with Bitonic Sort and then it repeats the experiment from Section 3.4.5, under the same testing conditions, to investigate the improvements made up to this point.

3.5.6.1 Nearly Sort vs Bitonic Sort

The two algorithms are compared by passing the same random input pair: **x** and **ncopies** which is again generated from `MVR` as in Section 3.4.5.1.

As we can see from Figure 3.21, Nearly Sort is significantly faster than the other algorithms and especially Bitonic Sort for a low number of cores. Then, when P increases the performance of both algorithms become closer because the achieved time complexity is asymptotically the same.

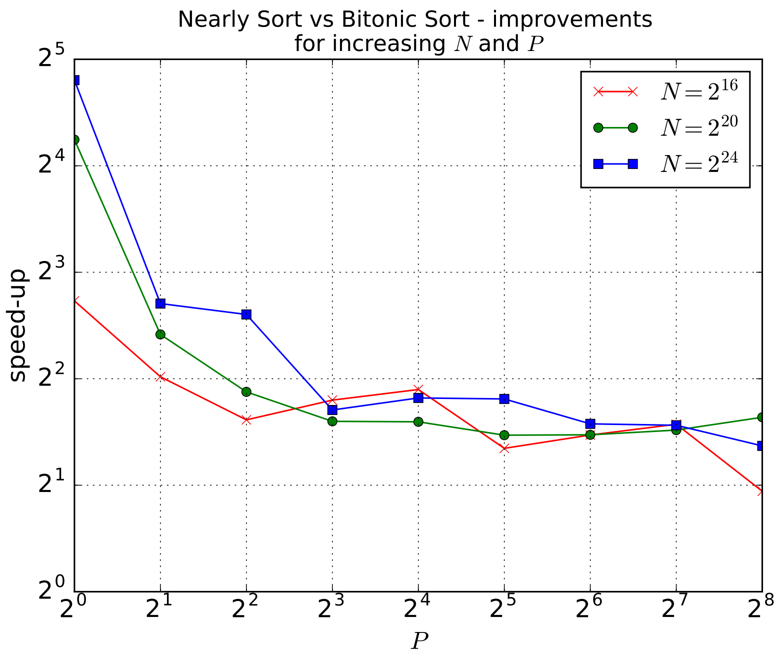
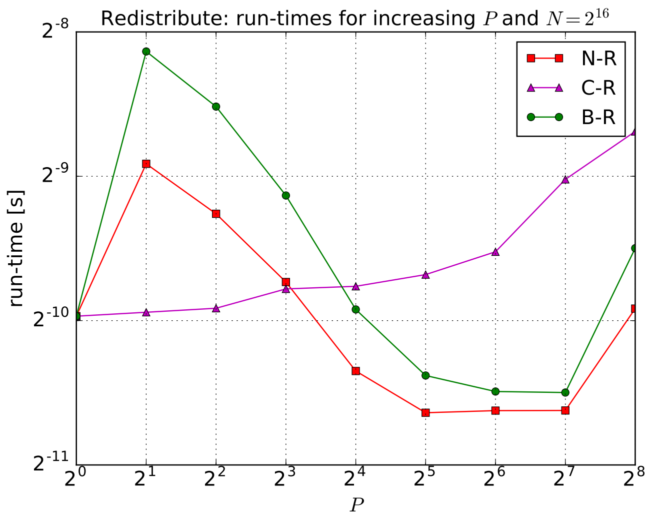


FIGURE 3.21: Nearly Sort vs Bitonic Sort - run-time speed-ups for increasing P

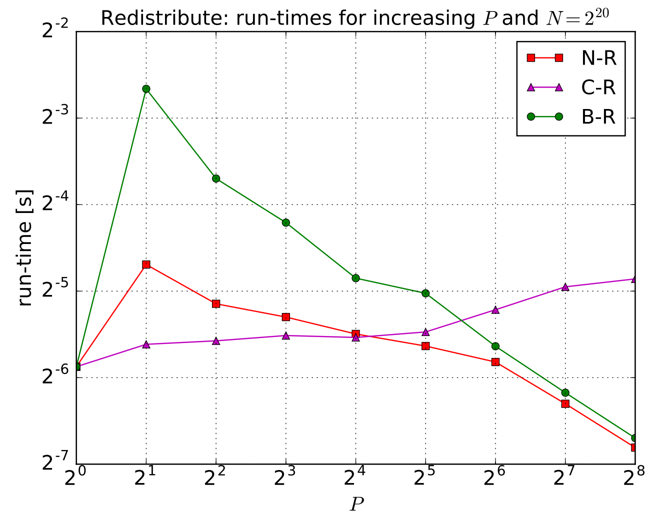
3.5.6.2 N-R vs B-R and C-R

In this experiment, the same strategy described in the previous section is used, since the required input for N-R, B-R and C-R is the same as for Bitonic Sort or Nearly Sort. The results are shown in Figure 3.22.

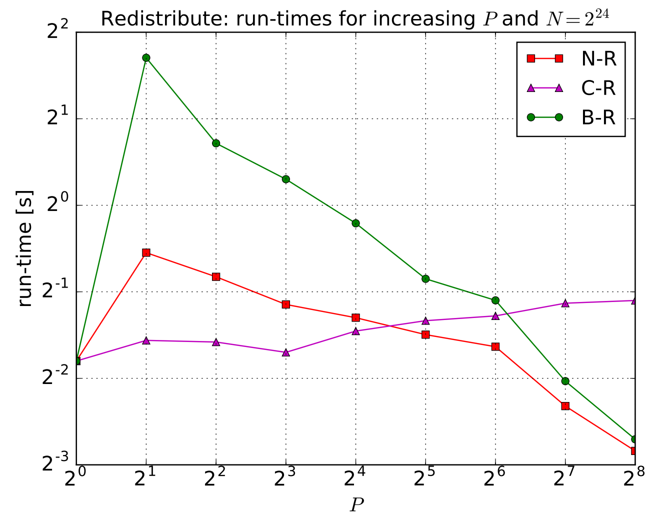
As expected from the previous results, and also from comparing (3.3) with (3.9), N-R is better than B-R overall: more precisely, it is much faster for a small number of cores, but comparable for high DOP. It is also interesting to see that N-R requires a lower DOP than N-R to outperform C-R.



(A) $N = 2^{16}$

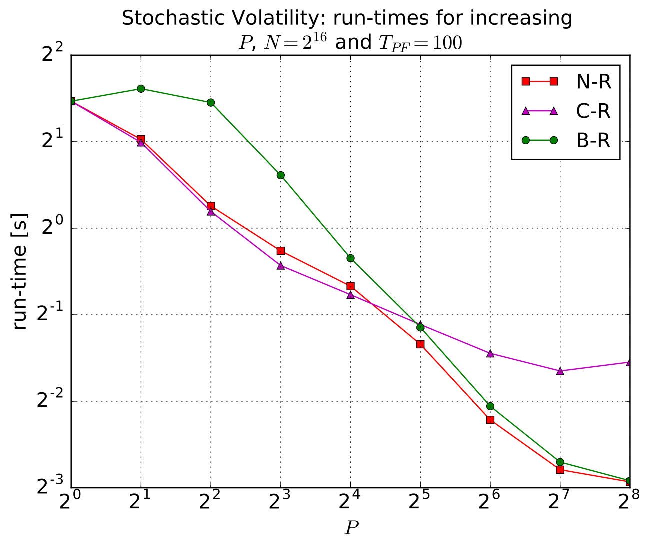


(B) $N = 2^{20}$

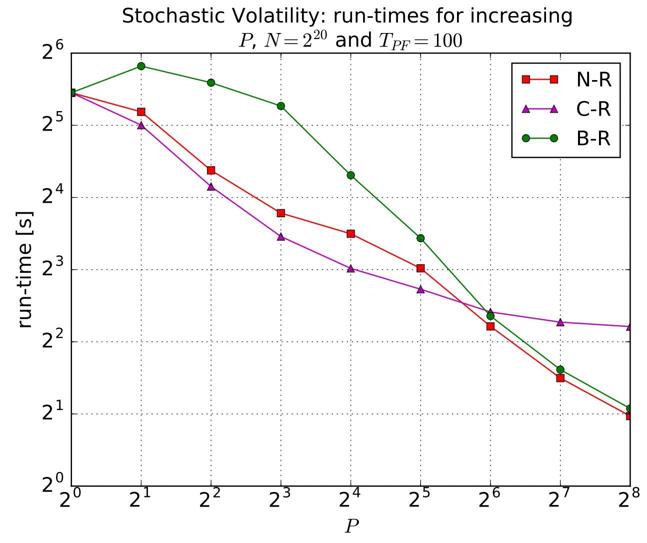


(C) $N = 2^{24}$

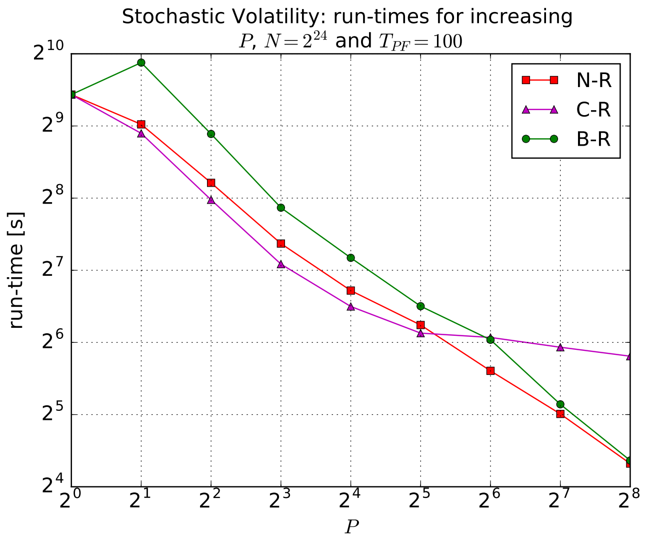
FIGURE 3.22: N-R vs B-R vs C-R - run-times for increasing N and P



(A) $N = 2^{16}$



(B) $N = 2^{20}$



(C) $N = 2^{24}$

FIGURE 3.23: Stochastic Volatility - run-times of PF with N-R or B-R or C-R for increasing N and P

3.5.6.3 Stochastic Volatility

This section repeats the experiment in Section 3.4.5 under the same testing conditions. Hence the stochastic volatility model is run again by three versions of Algorithm 1, only differing for the chosen redistribution, which in this case may be N-R, B-R or C-R.

We can observe in Figure 3.23 that a PF with N-R becomes faster than a PF with C-R for at least $P = 32$, as expected from the previous section, while a PF with B-R requires at least 64 cores to achieve the same result. For high DOP, using N-R instead of C-R makes the PF up to 3 times faster. On the other hand, the improvements that N-R brings over B-R can be mostly appreciated for low P, where switching Bitonic Sort with Nearly Sort speeds up the run-time by up to a factor of two. For the highest values of P, using N-R to parallelise redistribute in a PF provides about a 25% improvement vs the same PF using B-R as redistribute parallelisation.

3.6 Conclusions

This chapter has provided the first milestone of this thesis: proving that MPI is a much more suitable framework than MapReduce for B-R, the fully-balanced redistribute presented in [85]. The results provided in [85] show that B-R on MapReduce cannot outperform C-R even for $P = 512$ cores, while the results in this chapter prove that B-R on MPI can indeed outperform C-R by almost a three-time factor for 256 MPI cores.

In Section 3.5, it has also been shown that the performance can be further improved by switching Bitonic Sort, the most computationally intensive task in B-R, with Nearly Sort, a novel routine that does not perfectly sort the particles, but simply separates those that have to be duplicated from those that have to be deleted. The improvements vs B-R are significant for low DOP but incremental for high values of P.

These final results may sound disappointing. However, proving that sort is not necessary will be crucial for the novelty of the next chapter, where a novel fully-balanced redistribute for DMAs which achieves $O(\log_2 N)$ time complexity is presented.

Chapter 4

An $O(\log_2 N)$ Fully-Balanced Particle Filter for Distributed Memory Architectures

4.1 Introduction

B-R and N-R, the two redistributions in Algorithm 20 and 24, both presented in Chapter 3 and published in [89, 90], are significant performance bottlenecks to SMC methods on DMAs, as they both account for over 80% of the total run-time when the DOP is large. This is because every other task in SMC methods scales at most as $O(\log_2 N)$, while N-R and B-R achieve $O((\log_2 N)^2)$ time complexity due to four major contributing factors. The first is the sorting network in Figures 3.9 and 3.19, consisting of a bottom-up binary tree whose nodes are top-down binary trees themselves. In the second part of both algorithms, there is another top-down binary tree whose nodes are sequences of Cumulative Sum, Pivot Bcast, and rotational shifts, each taking $O(\log_2 N)$ computations. It is then clear that, until each of these four subtasks are improved to, or substituted by alternative strategies that scale logarithmically, redistribute will always be a largely intensive bottleneck.

The goal of this chapter is to completely re-design the redistribute in [89, 90] and propose a novel fully-balanced approach for DMAs which achieves $O(\log_2 N)$ time complexity. In doing so, the rest of this chapter is organised as follows: Section 4.2 discusses three crucial ideas to derive a $O(\log_2 N)$ Nearly Sort algorithm. Section 4.3 illustrates how to translate the same ideas to design a novel three-phase $O(\log_2 N)$ fully-balanced redistribution and show the improvements vs N-R and B-R. Section 4.4 proves that one of the phases of the novel $O(\log_2 N)$ redistribute can be entirely removed with little overhead. Here, the numerical results are provided for two exemplary PFs: the same stochastic volatility example described in Chapter 3 and a real-world model for alloy productions, in which a large N was shown to be necessary to meet the accuracy constraints. Section 4.5, outlines the conclusions of this chapter and introduces the goals for the next one.

4.2 Rotational Nearly Sort: An $O(\log_2 N)$ Nearly Sort

Sorting networks that scale as $O(\log_2 N)$ already exist. The first example is AKS sort, which can be found in [2]. The key idea is to organise the network as a top-down binary tree, where each node is designed as an *imperfect halver*, a circuit which splits the father node into two child nodes, each having a subset of wrong keys, called *strangers*, which should belong to other nodes. Each node is then tasked with computing the halver, and sending the strangers back to the father node.

However, the imperfect halver and the need of re-routing the strangers require such a large constant time c that AKS sort is slower than Bitonic Sort, unless for an impractically large N. The exact value of c is unknown as it depends on the network parameters, but is believed to be in the order of tens of thousands. In [70, 77], the original network has been simplified to have $c = 6100$ in the best parameter configuration. Nevertheless, the achieved constant is still too high: in [7], it has been estimated that a hypothetical $c = 87$ would require $N \geq 2^{173}$ keys to make AKS-like sorting networks faster than Bitonic Sort. Some other academics are of the opinion that the actual minimum N to observe a crossing point vs Bitonic Sort should be about 2^{78} [50], which is impractical anyway.

Although sorting in logarithmic time is not practically possible at the current state-of-the-art, similar concepts to AKS sort can be extrapolated to design a novel $O(\log_2 N)$ Nearly Sort with a practical constant time, as Nearly Sort is indeed a much easier property to achieve than sort.

4.2.1 Alternative Version of $O((\log_2 N)^2)$ Nearly Sort

The concept of strangers in AKS sort can be applied to Nearly Sort. The preliminary idea is to use a bottom-up binary tree structure where, at each stage, the nodes are divided pairwise into couples such that the coupled nodes, each containing a nearly-sorted sequence, are merged into a new father node which also becomes nearly-sorted. However, instead of merging the nodes as in Figure 3.19, the higher ranked child node could be tasked with moving its stranger keys to its sibling node.

As said in the previous section, the circuit of each node in AKS to identify and re-route the strangers has an impractical constant time. Alternatively, one could use a combination of Cumulative Sum and reduction to identify the strangers and their final destination in the memory of the sibling node. More precisely, let **zeros** be an array containing a “binary” Cumulative Sum that counts the number of $\mathbf{ncopies}^i = 0$ up to the i-th element within the memory of each node. Now, let $\bar{p}$ be lowest MPI rank of each node. It can be inferred that each i-th element of **zeros** within the node’s memory partition is computed as follows:

$$\mathbf{zeros}^i = \sum_{j=\bar{p}\times\frac{N}{P}}^{i} (1 - \text{sgn}(\mathbf{ncopies}^j)) \tag{4.1}$$

The i-th particle is then a stranger if the following boolean condition is true:

$$\mathbf{ncopies}^i > 0 \wedge \mathbf{zeros}^i > 0 \tag{4.2}$$

This corresponds to checking whether the i-th particle has to be duplicated and is in the wrong position, since some gaps are available.

At each stage k, the strangers (if there is any) must be moved by as many positions as:

$$\mathbf{r}^k = \sum_{j=0}^{\frac{\mathbf{N}_k}{\mathbf{P}_k}-1} (1 - \text{sgn}(\mathbf{ncopies}^j)) = \mathbf{zeros}^{\frac{\mathbf{N}_k}{\mathbf{P}_k}-1} \tag{4.3}$$

the number of zeros in the sibling node, which we know can be filled in with particles to copy. Here, it is clarified that $\mathbf{N}_k$ and $\mathbf{P}_k$ are the number of particles and cores per node during the k-th stage. The value of $\mathbf{r}^k$ is inferred from 4.1, and the core who owns it can broadcast it to the other cores in the node by using `MPI_Bcast` or `MPI_Allreduce` as in `Pivot_Bcast` (see Algorithm 18). The child nodes can then save all particles for which (4.2) is false, and use rotational shifts to move the others by $\mathbf{r}^k$ positions according to the bits of $\mathbf{r}^k$, from the LSB to the MSB. Then the saved particles are restored.

This implementation of Nearly Sort is a bottom-up equivalent version of the second part in B-R and N-R. But, if this alternative version is used in N-R in place of Algorithm 23, we can infer that now redistribute scales as $O((\log_2 N)^2)$ because of the recursive use of three factors instead of four: Cumulative Sum, reduction and rotational shifts.

4.2.2 One Cumulative Sum for All Pivots

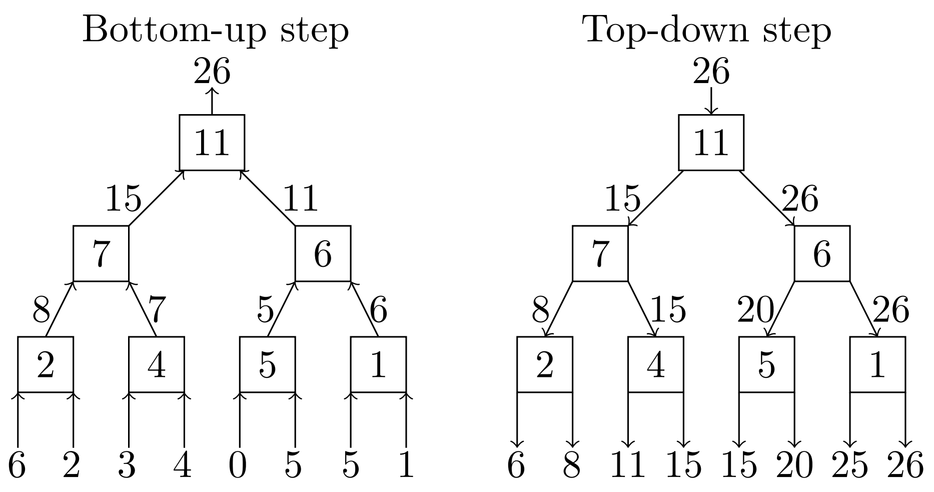


FIGURE 4.1: Parallel Cumulative Sum - original structure

Both Equations (4.2) and (4.3) depend on the same array **zeros**. This strongly suggests that the operation to identify the strangers and the rotations to perform could be one instead of two. Most importantly, it is possible to prove that the operation could be performed once instead of $\log_2 N$ times.

The parallel Cumulative Sum illustrated in Figure 3.5 is the most recent and efficient version. In the original implementation presented in [45], this operation is divided into a bottom-up and a top-down phase. During the bottom-up phase, the nodes compute Sum and in the meantime they build a tree where each node stores the value to subtract during the top-down phase (see Figure 4.1).

Therefore, the cores could build a tree containing the information of each $\mathbf{r}^k$ $\forall k = 1, 2, \dots \log_2 P$, while they compute **zeros**. Alternatively, the values of $\mathbf{r}^k$ can be updated stage by stage the same way, before every call of rotational shifts. Either way, the cost of computing all values in $\mathbf{r}$ is $O(\log_2 N)$ and the cost for each $\mathbf{r}^k$ is now $O(1)$. Therefore, Nearly Sort now scales as $O((\log_2 N)^2)$ only because of the repeated use of rotational shifts.

4.2.3 One Round of Rotational Shifts for All Particles

The alternative Nearly Sort described so far makes each particle rotate by a total number of positions equal to:

$$\mathbf{shifts}^i = \sum_{k=1}^{\log_2 P} \mathbf{r}^k \in \mathbb{Z} \tag{4.4}$$

At this point, by carefully profiling the performance of this alternative Nearly Sort, it is possible (although not trivial) to notice that the bits of $\mathbf{r}^k$ are most often 0 and very rarely 1, meaning that a lot of communication effort is wasted.

This communication inefficiency can be explained by pointing out that

$$\begin{aligned}\mathbf{shifts}^i &= \sum_{k=1}^{\log_2 P} \mathbf{r}^k = \sum_{k=1}^{\log_2 P} \sum_{j=0}^{\frac{\mathbf{N}_k}{\mathbf{P}_k}-1} (1 - \operatorname{sgn}(\mathbf{ncopies}^j)) \\ &= \sum_{j=0}^{i} (1 - \operatorname{sgn}(\mathbf{ncopies}^j)) \leq N - 1 \in \mathbb{Z} \end{aligned} \tag{4.5}$$

This means that not only each $\mathbf{r}^k$, but also the total number of zeros on the left of each i-th particle $\mathbf{shifts}^i$ can still be expressed in base-2 using up to $\log_2 N$ bits. Therefore, in order to Nearly Sort the particles, one could use Cumulative Sum once only to infer $\mathbf{shifts}^i$ and follow its bits to perform one round of rotational shifts, which both take $O(\log_2 N)$. Because the values in **shifts** are designed to place the particles to copy side by side, a particle cannot collide or get past another one. For clarity, these problems are formally defined a follows.

Definition 4.1. Let $\mathbf{x}^i$ be a particle having $\mathbf{ncopies}^i \geq 1$, and $\mathbf{x}^j$ be a particle having $\mathbf{ncopies}^j \geq 1$, with $j = i + dist$, where $0 < dist < N - 1$. A collision would occur if *dist* is a power-of-two number, $\mathbf{x}^j$ is rotating to the left by *dist*, and $\mathbf{x}^j$ is staying where it is. More formally, a collision occurs if the total number of rotations that $\mathbf{x}^j$ must perform has significant bit (i.e. the bit corresponding to *dist* rotations) equal to 1, while the same bit of the number of rotations that $\mathbf{x}^i$ must perform is 0. The same definition can be applied to collisions when rotations are performed to right if $\mathbf{x}^i$ is rotating to the right (and hence have significant bit equal to 1) and $\mathbf{x}^j$ is not rotating (and hence have significant bit equal to 0).

Definition 4.2. Let $\mathbf{x}^i$ be again a particle having $\mathbf{ncopies}^i \geq 1$, and $\mathbf{x}^j$ be again a particle having $\mathbf{ncopies}^j \geq 1$, with $j = i + dist$, where $0 < dist < N - 1$. The particle $\mathbf{x}^j$ can get past $\mathbf{x}^i$ if is rotating to the left by a power-of-two number greater than *dist* while $\mathbf{x}^i$ is staying where it is. The same problem occurs when rotations are performed to the right if $\mathbf{x}^i$ is rotating to the right by a power-of-two number greater than *dist* while $\mathbf{x}^j$ is staying where it is.

In the rest of this chapter, it is proven that these problems can never occur. Therefore, it is now possible to Nearly Sort N particles in logarithmic time complexity.

4.2.4 Rotational Nearly Sort

This section describes in technical details how to implement the ideas explained in the previous sections, in order to derive a $O(\log_2 N)$ alternative to Nearly Sort. The name that has been chosen for this algorithm is Rotational Nearly Sort.

As in Algorithm 22, the first thing to do is to Nearly Sort the particles locally by calling S-NS as this routine only takes $O(\frac{N}{P})$ iterations. This will start moving the particles locally to the left which is what it has to be done across the MPI cores eventually. At this point, the particles within core p must shift to the left by as many positions as the number of zero elements in **ncopies** owned by the cores with a lower rank. Let $\mathbf{zeros} \in \mathbb{Z}^P$ be the array which counts the number of $\mathbf{ncopies}^i = 0$ within each core; each element of $\mathbf{shifts} \in \mathbb{Z}^P$ (the array to keep track of the remaining shifts) can be initialised as follows:

$$\mathbf{shifts}^p = \sum_{\tilde{p}=0}^{p-1} \mathbf{zeros}^{\tilde{p}} \tag{4.6}$$

As anticipated in Section 4.2.2, Equation (4.6) can be parallelised by using parallel exclusive Cumulative Sum once, after each core p has initialised $\mathbf{zeros}^p$ to the sum of zeros within its memory, at the end of S-NS. Then the particles can be rotated by following the information stored in **shifts**.

As explained qualitatively in Section 4.2.3, it is now necessary to express $\mathbf{shifts}^p$ in binary notation and shift the particles by increasing power-of-two numbers of positions, depending on the bits of $\mathbf{shifts}^p$, from the Least Significant Bit (LSB) to the Most Significant Bit (MSB). This translates to using a bottom-up binary tree structure which has depth $O(\log_2 N)$ as long as Equation (4.6) is updated in constant time. Further technical details follow.

If $P < N$, then Rotational Nearly Sort performs first an extra leaf stage in which the MPI cores send all the particles to the neighbor on their left, if the bitwise & of $\mathbf{shifts}^p$ and $\frac{N}{P}-1$ is positive. In addition, they also send $\mathbf{shifts}^p - \mathbf{shifts}^p \& \left(\frac{N}{P}-1\right)$, the remaining shifts to perform after the leaf stage. In simple terms, the leaf stage masks the $\log_2 \frac{N}{P}$ LSBs of $\mathbf{shifts}^p$ and performs at once of the rotations referring to those bits.

After the leaf stage, the actual tree-structure routine can start. At every k-th stage of the tree (for $k = 1, 2, \dots, \log_2 P$), any core p will send to its partner $p - 2^{k-1}$ all its particles (i.e. $\mathbf{x}$ and $\mathbf{ncopies}$) and $\mathbf{shifts}^p - \frac{N}{P}2^{k-1}$ (i.e. the number of remaining shifts after the current rotation) if and only if the bitwise & of $\mathbf{shifts}^p$ and $\frac{N}{P}2^{k-1}$ is positive; this corresponds to checking a new bit of $\mathbf{shifts}^p$, precisely the one which is significant at the current stage. At every stage, the particles will then shift by an increasing power-of-two number of positions and $\mathbf{shifts}$ gets updated in $O(1)$. Therefore, since $\mathbf{shifts}^p \leq N-1$ (because a particle must shift at most from one end to the other end) and its value is updated in constant time, the overall achieved time complexity is equal to $O(\log_2 N)$. The following theorem and corollary prove that a particle can never collide with or get past another one.

Theorem 4.3. *During the k-th iteration of Rotational Nearly Sort, $\forall k = 1, 2, \dots, \log_2 P$, a particle $\mathbf{x}^j$, having $\mathbf{ncopies}^j \geq 1$ and rotating to the left by $\frac{N}{P}2^{k-1}$ positions, can never collide with a particle $\mathbf{x}^i$, having $\mathbf{ncopies}^i \geq 1$ and $j = i + \frac{N}{P}2^{k-1}$.*

Proof of Theorem 4.3. At the k-th iteration, particle $\mathbf{x}^i$ has $\mathbf{shifts}^i$ remaining rotations to the left, while $\mathbf{x}^j$ has $\mathbf{shifts}^j$. Therefore, the necessary and sufficient condition for collisions, defined in Definition 4.1, can be restated for Rotational Nearly Sort by checking whether the following logical condition

$$\left(\left(\mathbf{shifts}^i \& \frac{N}{P}2^{k-1}\right) = 0\right) \wedge \left(\left(\mathbf{shifts}^j \& \frac{N}{P}2^{k-1}\right) > 0\right) \tag{4.7}$$

is true, which corresponds to checking whether the significant bit of $\mathbf{shifts}^i$ is 0 and the significant bit of $\mathbf{shifts}^j$ is 1. This condition can also be rearranged as follows:

$$\left(\left(\mathbf{shifts}^i \& \frac{N}{P}2^{k-1}\right) = 0\right) \wedge \left(\left((\mathbf{shifts}^i + \mathbf{zeros}^{i+1:j-1}) \& \frac{N}{P}2^{k-1}\right) > 0\right) \tag{4.8}$$

where $\mathbf{zeros}^{i+1:j-1}$ is the number of 0s in $\mathbf{ncopies}$ between positions i and j excluded.

Since Rotational Nearly Sort performs rotations using a LSB-to-MSB strategy, it is easy to infer that the bits to the right of the significant one at this iteration (i.e. bit $k-1$) are all 0. This means that, if the significant bit of $\mathbf{shifts}^i$ is 0, the only condition that would make (4.8) true would be $\mathbf{zeros}^{i+1:j-1} = \frac{N}{P}2^{k-1}$. That is, however, impossible because in this case there are only $i + \frac{N}{P}2^{k-1} - 1$ memory slots between i and j, which means that:

$$\mathbf{zeros}^{i+1:j-1} \leq j - i - 1 = \frac{N}{P}2^{k-1} - 1 < \frac{N}{P}2^{k-1} \tag{4.9}$$

□

Corollary 4.4. *During the k-th iteration of Rotational Nearly Sort, $\forall k = 1, 2, \dots, \log_2 P$, a particle $\mathbf{x}^j$, having $\mathbf{ncopies}^j \geq 1$ and rotating to the left by $\frac{N}{P}2^{k-1}$ positions, can never get past a particle $\mathbf{x}^i$, having $\mathbf{ncopies}^i \geq 1$ and $i < j < i + \frac{N}{P}2^{k-1}$.*

Proof of Corollary 4.4. Theorem 4.3 can automatically prove Corollary 4.4 because, once again, (4.8) is true only if $\mathbf{zeros}^{i+1:j-1} = \frac{N}{P}2^{k-1}$. But in this case $j < i + \frac{N}{P}2^{k-1}$, and hence:

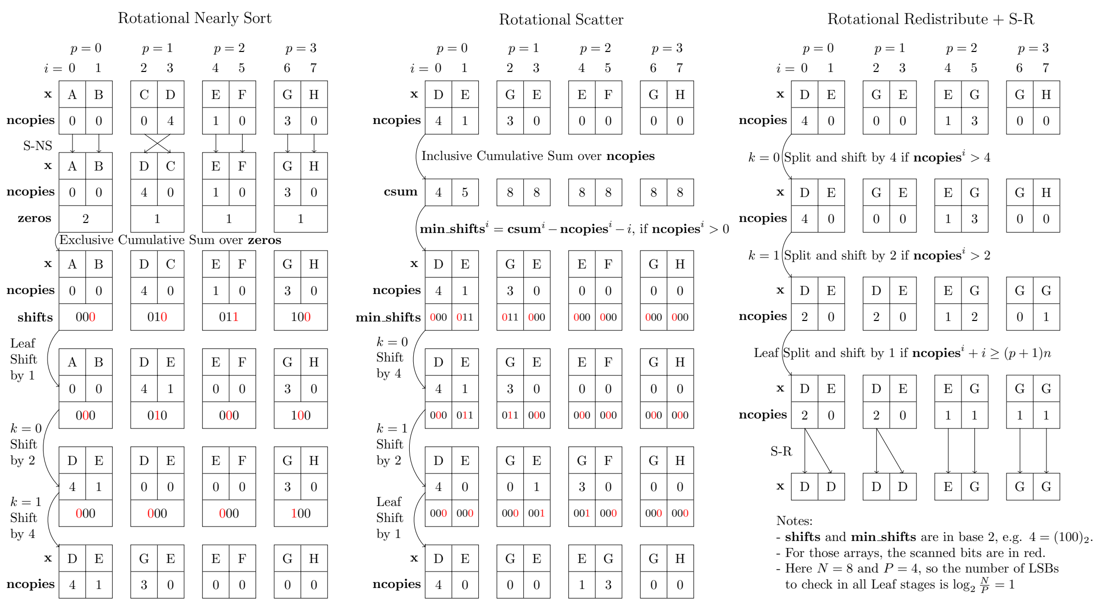
$$\mathbf{zeros}^{i+1:j-1} \leq j - i - 1 < j - i < \frac{N}{P}2^{k-1} \tag{4.10}$$

□

In simpler words, Theorem 4.3 and Corollary 4.4 reinforce the following statement: all non-zero values in **shifts** are by definition monotonically increasing (see Equation (4.4)), and because a LSB-to-MSB strategy is applied, a particle rotating to the left can at most catch up with the next one and there are always enough zeros to its left to do that safely.

This breakthrough is very encouraging. The following section applies similar concepts to develop a $O(\log_2 N)$ fully-balanced redistribute for DMAs and offers some algorithmic implementation details about each step (including Rotational Nearly Sort).

4.3 A Three Step $O(\log_2 N)$ Fully-Balanced Redistribute for DMAs



Rotational Nearly Sort
Rotational Scatter
Rotational Redistribute + S-R
p = 0, p = 1, p = 2, p = 3
i = 0 1 2 3 4 5 6 7
x
ncopies
S-NS
zeros
Exclusive Cumulative Sum over zeros
shifts
Leaf Shift by 1
k = 0 Shift by 2
k = 1 Shift by 4
Inclusive Cumulative Sum over ncopies
csum
min_shifts^i = csum^i − ncopies^i − i, if ncopies^i > 0
min_shifts
k = 0 Shift by 4
k = 1 Shift by 2
Leaf Shift by 1
k = 0 Split and shift by 4 if ncopies^i > 4
k = 1 Split and shift by 2 if ncopies^i > 2
Leaf Split and shift by 1 if ncopies^i + i ≥ (p + 1)n
S-R
Notes:
- shifts and min_shifts are in base 2, e.g. 4 = (100)_2.
- For those arrays, the scanned bits are in red.
- Here N = 8 and P = 4, so the number of LSBs to check in all Leaf stages is log_2 N/P = 1

FIGURE 4.2: $O(\log_2 N)$ redistribute - example for $N = 8$ and $P = 4$

This section describes how it is possible to extrapolate the concepts in Rotational Nearly Sort to develop a novel fully-balanced redistribution for DMAs, which serves its purpose by using a three-phase approach, each phase taking $O(\log_2 N)$ computations.

The reader is referred to Figure 4.2 which illustrates an example for $N = 8$ particles and $P = 4$ cores. In the first phase, this routine moves to the left all particles that must be duplicated by using Rotational Nearly Sort. The goal of the second phase, called Rotational Scatter, is to create gaps to safely duplicate those particles without risking collisions. The last phase, Rotational Redistribute, fills the gaps by duplicating the particles across the MPI cores. This way, (3.2) is achieved. If $P < N$, then Rotational Redistribute is finalised by S-R to redistribute the particles within each MPI cores.

The following two sections describe Rotational Scatter and Rotational Redistribute while Section 4.3.5 repeats the experiment of Section 3.5.6 to show the improvements that this novel redistribute provides vs B-R and N-R.

4.3.1 Rotational Scatter

The goal of this phase is to make room for each particle that has to be copied. We know that after Rotational Nearly Sort, the first $1 < m \leq N$ elements in **ncopies** will be positive (see (3.8)). We also know that Equation (2.17) always guarantees that for every $\mathbf{ncopies}^i > 0$ there are as many zeros as $\mathbf{ncopies}^i - 1$. Therefore, making room for the copies to create easily translates to shifting the particles to the right until **ncopies** has the following new shape:

$$\mathbf{ncopies} = [\lambda^0, 0, \dots, 0, \lambda^1, 0, \dots, 0, \lambda^{m-1}, 0, \dots, 0] \tag{4.11}$$

where for each $\mathbf{ncopies}^i > 0$ (generically represented by λs in (4.11)), $\mathbf{ncopies}^i - 1$ zeros follow. Let $\mathbf{csum} \in \mathbb{Z}^N$ be the inclusive Cumulative Sum of **ncopies**. To achieve (4.11), it can be inferred that for each index i such that $\mathbf{ncopies}^i > 0$, the minimum required number of shifts to the right that the i-th particle must perform is:

$$\begin{aligned}\mathbf{min\_shifts}^i &= \sum\nolimits_{j=0}^{i-1} (\mathbf{ncopies}^j - 1) \\ &= \mathbf{csum}^i - \mathbf{ncopies}^i - i\end{aligned} \tag{4.12}$$

since the i-th element in the inclusive Cumulative Sum is $\mathbf{csum}^i = \sum_{j=0}^{i} \mathbf{ncopies}^j$.

As in Rotational Nearly Sort, parallel inclusive Cumulative Sum is computed once only and, after that, Equation (4.12) is trivially parallelisable. However, in this phase $\mathbf{min\_shifts} \in \mathbb{Z}^N$, i.e. each core now owns $n = \frac{N}{P}$ elements of **min_shifts** instead of one, which is the case with **shifts** in Rotational Nearly Sort.

In this case too, each $\mathbf{min\_shifts}^i$ is expressed in base-2. However, here the bits are scanned from the MSB to the LSB, as the goal is now to scatter the particles, instead of gathering them together. Each particle is then rotated to the right by decreasing power-of-two numbers of position, depending on the value of the scanned bits. This corresponds to using a top-down binary tree structure which, once again, takes $O(\log_2 N)$ steps as long as (4.12) can be updated in constant time. Technical details follow.

At each stage k of the tree, any core with rank p will send to its partner with rank $p + \frac{P}{2^{k+1}}$ any particle i for which the bitwise & of $\mathbf{min\_shifts}^i$ and $N2^{-k}$ is positive and will hold the other particles. This corresponds to scanning a new bit of $\mathbf{min\_shifts}^i$, the one which is significant to the k-th stage. In case $P < N$, after $\log_2 P$ stages the cores will perform a leaf stage to send at once to their neighbour with higher rank any particle i such that $\mathbf{min\_shifts}^i + i > (p+1)n$, which corresponds to checking whether the i-th particle should be placed in the memory partition of the neighbour core. In other words, the leaf stage performs at once the rotations corresponding to the $\log_2 \frac{N}{P}$ least significant bits of each $\mathbf{min\_shifts}^i$.

In order to ensure logarithmic time complexity, one needs to update **csum** and **min_shifts** in $O(\frac{N}{P})$. This can be done if the cores send $starter = \mathbf{csum}^j$, where j is the index of the first particle to send if there is any to send. It is possible to prove that a particle can never collide with or get past another one. This statement will be proven later in this section. Hence, each core can safely see the received $starter$ as $\sum \mathbf{ncopies}$ for the cores with lower rank, and use it to re-initialise and update **csum** sequentially as in (3.1). This strategy guarantees **csum** is always correct for at least any index i such that $\mathbf{ncopies}^i > 0$, but those are the only indexes of interest. After updating **csum**, **min_shifts** can be recomputed in $O(\frac{N}{P})$ by using Equation (4.12).

It is easy to infer that $\mathbf{min\_shifts}^i < N - 1$, for any $0 \leq i \leq N-1$, as a particle could be shifted at most from the second to the last position. Therefore, because the shifts will decrease stage by stage by up to a factor of two, the achieved time complexity of Rotational Scatter is $O(\log_2 N)$. The following theorem proves that two particles can never collide or get past each other.

Theorem 4.5. *Given a nearly-sorted input* **ncopies***, at the k-th iteration of Rotational Scatter, $\forall k = 1, 2, \dots, \log_2 P$, a particle $\mathbf{x}^i$, having* $\mathbf{ncopies}^i \geq 1$ *and rotating to the right by $N2^{-k}$ positions, can never collide with or get past a particle $\mathbf{x}^j$, having* $\mathbf{ncopies}^j \geq 1$ *and $j = i + dist$ with $1 \leq dist \leq N2^{-k}$.*

Proof of Theorem 4.5. This theorem can be proved in two possible complementary cases:

1. there is one or more zeros between i and j;

2. there are no zeros between i and j.

Case 1. Since the particles are initially nearly-sorted, at the beginning there are no zeros in between any pair of particles in position i and j. At the k-th iteration, if one or more zeros is found between $\mathbf{x}^i$ and $\mathbf{x}^j$, it necessarily means that $dist > \mathbf{min\_shifts}^i \geq N2^{-k}$. That is because of two reasons. First, zeros between two particles $\mathbf{x}^i$ and $\mathbf{x}^j$ can only be created if the MSB of $\mathbf{min\_shifts}^i$ is 0 and the MSB of $\mathbf{min\_shifts}^j$ is 1. Second, for any binary number, its MSB, if equal to 1, is a greater number than the one represented by any disposition of all remaining LSBs (e.g. $(1000)_2 = 8 > (0111)_2 = 7$). Hence, if there is any zero between $\mathbf{x}^i$ and $\mathbf{x}^j$, it is because during at least one of the previous iterations, $\mathbf{x}^j$ rotated by an MSB and $\mathbf{x}^i$ did not, such that $\mathbf{x}^j$ is now beyond reach of possible collisions with $\mathbf{x}^i$.

Case 2. In this case, all particles between i and j are still nearly-sorted. Therefore, $\mathbf{x}^i$ would collide with (when $dist = N2^{-k}$) or get past $\mathbf{x}^j$ (when $dist < N2^{-k}$) if

$$\Big(\Big(\textbf{min\_shifts}^i \& N2^{-k}\Big) > 0\Big) \wedge \Big(\Big(\textbf{min\_shifts}^j \& N2^{-k}\Big) = 0\Big) \tag{4.13}$$

is true, which corresponds to checking whether the MSB of $\textbf{min\_shifts}^i$ is 1 and the MSB of $\textbf{min\_shifts}^j$ is 0. In other words, (4.13) can be simplified to checking if $\textbf{min\_shifts}^i > \textbf{min\_shifts}^j$. However, for a pair of particles $\mathbf{x}^i$ and $\mathbf{x}^j$ within a nearly-sorted group of particles, that is impossible because:

$$\begin{aligned}
\textbf{min\_shifts}^i &= \textbf{csum}^i - \textbf{ncopies}^i - i \\
&= \textbf{csum}^j - \sum\nolimits_{z=i+1}^{j} \textbf{ncopies}^z - \textbf{ncopies}^i - j + dist \\
&= \textbf{csum}^j - \textbf{ncopies}^j - j - \Big(\sum\nolimits_{z=i}^{j-1} \textbf{ncopies}^z - dist\Big) \\
&= \textbf{min\_shifts}^j - \Big(\sum\nolimits_{z=i}^{j-1} \textbf{ncopies}^z - dist\Big) \leq \textbf{min\_shifts}^j
\end{aligned}$$

since $\textbf{csum}^j = \sum_{z=0}^{j} \textbf{ncopies}^z$, $dist = j - i$ and (in this case) $\sum_{z=i}^{j-1} \textbf{ncopies}^z \geq j - i$. □

In simpler words, Theorem 4.5 reinforces the following statement: the particles are initially nearly-sorted; during the rotations, two particles can either be separated by enough zeros to never cause collisions (because of the MSB-to-LSB strategy) or be separated by nearly-sorted non-zero particles. For these adjacent particles, **min_shifts** is monotonically increasing and hence those particles can never collide with or get past each other, as the increasing values in **min_shifts**, along with the MSB-to-LSB strategy, can only make them scatter.

4.3.2 Rotational Redistribute

After Rotational Scatter, every particle such that $\textbf{ncopies}^i > 0$ has enough room on its right to duplicate itself $\textbf{ncopies}^i - 1$ times. The goal in this third and final phase, Rotational Redistribute, is to fill in the available gaps to balance the workload.

This is done once again by moving the particles to the right, as in Rotational Scatter. However, in this case there is no need to compute and update Cumulative Sum to infer the shifts to perform, because **ncopies** already carries all information required. Therefore, the strategy is to first express each $\textbf{ncopies}^i$ in binary notation and scan the bits of $\textbf{ncopies}^i$ from the MSB to the LSB. The bits still represent power-of-two numbers of positions which the particles might have to rotate by, depending on the value of the scanned bits. This translates again to using a top-down binary tree structure where the cores are connected as in Rotational Scatter.

At each k-th stage of the tree, every $\textbf{ncopies}^i$ can be written as follows:

$$\textbf{ncopies}^i = (\textbf{ncopies}^i - N2^{-k}) + N2^{-k} \tag{4.14}$$

Stage by stage, the cores need to split every $\mathbf{ncopies}^i > N2^{-k}$ by sending $\mathbf{ncopies}^i - N2^{-k}$ copies to its partner and keeping $N2^{-k}$. Checking $\mathbf{ncopies}^i > N2^{-k}$ is indeed equivalent to reading the bit of $\mathbf{ncopies}^i$ which is relevant to the k-th stage. Here it is specified that the significant bit is also 1 when $\mathbf{ncopies}^i = N2^{-k}$, but in that case $\mathbf{ncopies}^i - N2^{-k} = 0$ copies would be sent. Hence, checking whether $\mathbf{ncopies}^i > N2^{-k}$ is true is more informative than computing the bitwise & of $\mathbf{ncopies}^i$ and $N2^{-k}$.

If $P < N$, after $\log_2 P$ steps the cores need to perform a leaf stage to split any particle i for which $\mathbf{ncopies}^i + i \geq (p+1)n$, where $(p+1)n$ is the beginning of the partition which belongs to the neighbour core with rank $p+1$. This done by sending $\mathbf{ncopies}^i + i - (p+1)n$ copies (the ones in excess) to their neighbour and keeping the remaining $(p+1)n - i$ ones. Once again, the leaf stage performs at once all rotations that refer to the least $\log_2 \frac{N}{P}$ bits of the integer representing the shifts or the splits and shifts to perform, in this case $\mathbf{ncopies}^i$. After that, it is guaranteed that **ncopies** will still have shape (4.11) but, most importantly, the workload is fully-balanced across the MPI ranks, i.e. the sum of **ncopies** within each core's memory is equal to $\frac{N}{P}$.

Because Equation (2.17) holds, $\mathbf{ncopies}^i \leq N\ \forall i$ and because every integer number N can be expressed as a sum of $\log_2 N$ power-of-two numbers, we can infer Rotational Redistribute achieves $O(\log_2 N)$ time complexity. Also, collisions cannot occur in this step because (4.11) holds from the beginning. A proof of this statement is trivial and omitted for brevity.

4.3.3 $O(\log_2 N)$ Fully-Balanced Redistribute

After Rotational Redistribute, S-R can be invoked locally and will finish in $O(\frac{N}{P})$ iterations because the particles are now equally distributed across the MPI ranks.

The overall redistribute will then perform in sequence Rotational Nearly Sort, Rotational Scatter and Rotational Redistribute only if $P > 1$ before calling S-R. We can infer that the achieved time complexity is then $O(N)$ for $P = 1$, $O(\log_2 N)$ for $P = N$ cores and for any $1 \leq P \leq N$ is:

$$O\left(\frac{N}{P} + \frac{N}{P}\log_2 P\right) \tag{4.15}$$

The first term in (4.15) represents S-R, which is performed always, and all the steps which are called once only for any $P > 1$, such as S-NS. The second term describes the $\log_2 P$ stages of Rotational Nearly Sort, Rotational Scatter and Rotational Redistribute during which up to $\frac{N}{P}$ particles are updated, sent and received.

4.3.4 Algorithmic Implementation

This section provides algorithmic implementation details about Rotational Nearly Sort (see Algorithm 25), Rotational Scatter (see Algorithm 26), Rotational Redistribute (see Algorithm 27), and the $O(\log_2 N)$ fully-balanced redistribute described in the previous section (see Algorithm 28).

Algorithm 25 Rotational Nearly Sort

```
Input: x, ncopies, $N$, $P$, $n = \frac{N}{P}$, $p$
Output: x, ncopies
 1: x, ncopies, zeros ← S-NS(x, ncopies, $n$), see Algorithm 22
 2: shifts ← Exclusive_Cumulative_Sum(zeros)
 3: if $P < N$ then perform leaf stage of the binary tree
 4:     partner ← $(p-1)$ & $(P-1)$, i.e. the neighbour
 5:     if shifts & $(n-1) > 0$ then
 6:         for $j \leftarrow 0; j < n; j \leftarrow j+1$ do
 7:             if $j <$ shifts & $n-1$ then
 8:                 Send $\mathbf{x}^j$, $\mathbf{ncopies}^j$ to partner, $\mathbf{ncopies}^j \leftarrow 0$
 9:             else
10:                 Shift particle to the left by shifts & $n-1$
11:             end if
12:         end for
13:         shifts ← shifts − shifts & $n-1$
14:         Send shifts to partner
15:     else
16:         Send arrays of 0s to partner (Message to reject)
17:     end if
18:     Accept or reject the received particles and shifts
19: end if
20: for $k \leftarrow 1; k \leq \log_2 P; k \leftarrow k+1$ do binary tree
21:     partner ← $(p - 2^{k-1})$ & $(P-1)$
22:     if shifts & $n2^{k-1} > 0$ then
23:         for $j \leftarrow 0; j < n; j \leftarrow j+1$ do
24:             Send $\mathbf{x}^j$, $\mathbf{ncopies}^j$ to partner, $\mathbf{ncopies}^j \leftarrow 0$
25:         end for
26:         shifts ← shifts − shifts & $n2^{k-1}$
27:         Send shifts to partner
28:     else
29:         Send arrays of 0s to partner (Message to reject)
30:     end if
31:     Accept or reject the received particles and shifts
32: end for
```

Algorithms 25, 26 and 27 refer to the pseudo-code that a generic core p would run on a DMA (e.g. by using MPI). For this reason, the for loops, the particles and every array related to them are indexed using local indexes $0 \leq j \leq n-1$, where $n = \frac{N}{P}$, although all equations in the previous sections have been expressed in global index form (for brevity and simplicity reasons). Therefore, some local indexes in certain equations have to be rescaled by $p \times n$ to be converted to global indexes. This is because a local index j is equivalent to global index $i = j + p \times n$.

4.3.5 Numerical Results

This section repeats the same experiment from Section 3.4.5, under the same testing conditions and strategy. To do that, Algorithm 28 is first compared to N-R and B-R,

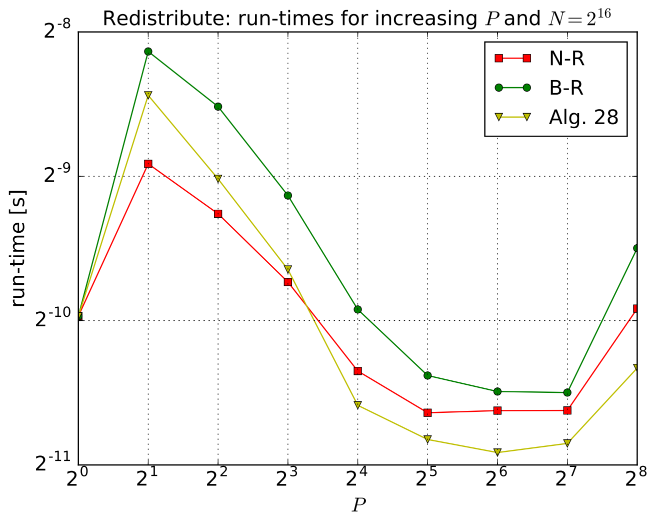
Algorithm 26 Rotational Scatter

```
Input: $\mathbf{x}$, $\mathbf{ncopies}$, $N$, $P$, $n = \frac{N}{P}$, $p$
Output: $\mathbf{x}$, $\mathbf{ncopies}$
1: $\mathbf{csum} \leftarrow \texttt{MPI\_Cumulative\_Sum}(N, P, \mathbf{ncopies})$
2: $\mathbf{min\_shifts}^j \leftarrow \mathbf{csum}^j - \mathbf{ncopies}^j - j - np$, $\forall j = 0, 1, \dots n-1$ if $\mathbf{ncopies}^j > 0$
3: for $k \leftarrow 0$; $k < \log_2 P$; $k \leftarrow k+1$ do Binary tree
4:     $partner \leftarrow \left(p + \frac{P}{2^{k+1}}\right) \& (P-1)$
5:     for $j \leftarrow 0$; $j < n$; $j \leftarrow j+1$ do
6:         if $\mathbf{min\_shift}^j > N2^{-k}$ then
7:             Send $\mathbf{x}^j$, $\mathbf{ncopies}^j$, $\mathbf{min\_shifts}^j - N2^{-k}$ to $partner$
8:             Send also $starter = \mathbf{csum}^j$ if $j$ is the first index to send
9:         else
10:            Send 0 to $partner$ (Message to reject)
11:        end if
12:    end for
13:    Accept or reject the received particles and $starter$
14:    Update $\mathbf{csum}$ sequentially if $starter$ has been received
15:    Update $\mathbf{min\_shifts}$ as in line 2
16: end for
17: if $P < N$ then Perform Leaf stage of the binary tree
18:    for $j \leftarrow 0$; $j < n$; $j \leftarrow j+1$ do
19:        if $\mathbf{min\_shifts}^j + j > (p+1)n$ then
20:            Send $\mathbf{x}^j$, $\mathbf{ncopies}^j$ to $partner$
21:        else
22:            Send 0 to $partner$ (Message to reject)
23:        end if
24:        if $\mathbf{min\_shifts}^j > 0$ then
25:            Shift particle to the right by $\mathbf{min\_shifts}^j$
26:        end if
27:    end for
28:    Accept or reject the received particles
29: end if
```

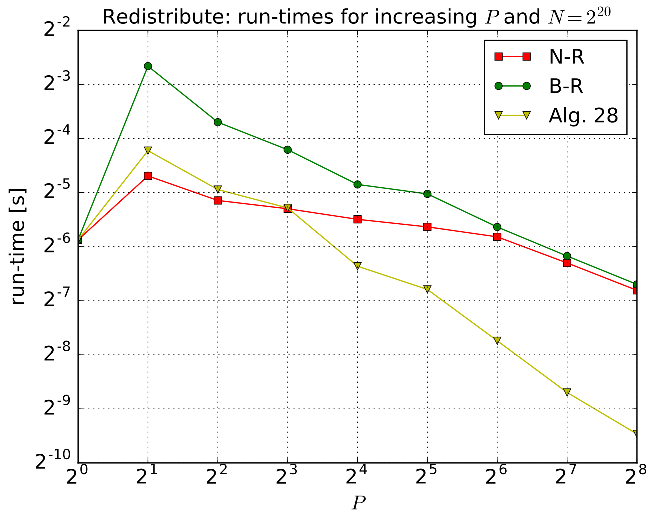
the two most advanced fully-balanced redistribute from Chapter 3, and then this section studies its impact on the same stochastic volatility example.

4.3.6 $O(\log_2 N)$ Redistribute vs B-R and N-R

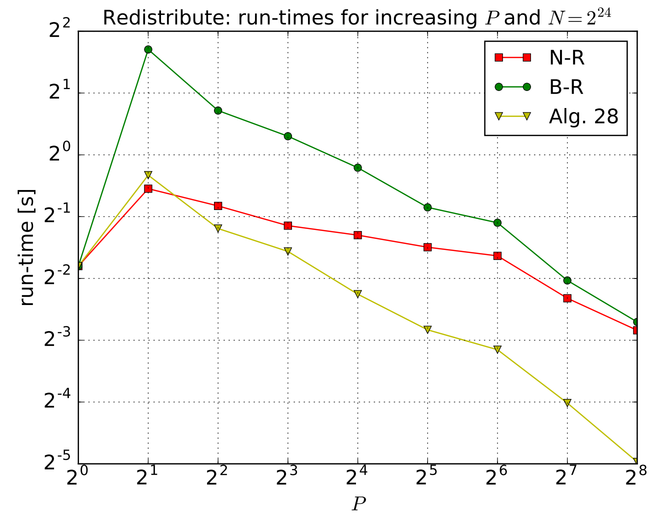
The results for this experiment are found in Figure 4.3. For $P = 2$, we can see that the $O(\log_2 N)$ redistribute is slightly worse than N-R, because in that case N-R sends two messages while Algorithm 28 sends three, but is also roughly three to four times as fast as B-R, due to the high volume of computation in Bitonic Sort. However, we can observe that the gap between Algorithm 28 and N-R and B-R tends to significantly increase with P, as the novel approach is on a faster scalability curve. For the highest number of cores $P = 256$, the $O(\log_2 N)$ redistribute becomes up to seven times faster than both N-R and B-R, depending on N.



(A) $N = 2^{16}$



(B) $N = 2^{20}$



(C) $N = 2^{24}$

FIGURE 4.3: $O(\log_2 N)$ redistribute vs B-R vs N-R - run-times for increasing N and P

Algorithm 27 Rotational Redistribute

```
Input: $\mathbf{x}$, $\mathbf{ncopies}$, $N$, $P$, $n = \frac{N}{P}$, $p$
Output: $\mathbf{x}$, $\mathbf{ncopies}$
 1: for $k \leftarrow 0$; $k < \log_2 P$; $k \leftarrow k + 1$ do Binary tree
 2:     $partner \leftarrow \left(p + \frac{P}{2^{k+1}}\right) \& (P - 1)$
 3:     for $j \leftarrow 0$; $j < n$; $j \leftarrow j + 1$ do
 4:         if $\mathbf{ncopies}^j > N2^{-k}$ then
 5:             Send $\mathbf{x}^j$, $\mathbf{ncopies}^j - N2^{-k}$ to $partner$
 6:             $\mathbf{ncopies}^j \leftarrow N2^{-k}$
 7:         else
 8:             Send 0 to $partner$ (Message to reject)
 9:         end if
10:     end for
11:     Accept or reject the received particles
12: end for
13: if $P < N$ then Perform Leaf stage of the binary tree
14:     for $j \leftarrow 0$; $j < n$; $j \leftarrow j + 1$ do
15:         if $\mathbf{ncopies}^j + j > (p + 1)n$ then
16:             Send $\mathbf{x}^j$, $\mathbf{ncopies}^j + j - (p + 1)n$ to $partner$
17:             $\mathbf{ncopies}^j \leftarrow (p + 1)n - j$
18:         else
19:             Send 0 to $partner$ (Message to reject)
20:         end if
21:     end for
22:     Accept or reject the received particles
23: end if
```

Algorithm 28 $O(\log_2 N)$ redistribute

```
Input: $\mathbf{x}$, $\mathbf{ncopies}$, $N$, $P$, $n = \frac{N}{P}$, $p$
Output: $\mathbf{x}$
1: if $P > 1$ then
2:     $\mathbf{x}, \mathbf{ncopies} \leftarrow$ Rot.-NS($\mathbf{x}, \mathbf{ncopies}, N, P, n, p$), $\mathbf{ncopies}$ has now shape (3.8)
3:     $\mathbf{x}, \mathbf{ncopies} \leftarrow$ Rot.-Scatter($\mathbf{x}, \mathbf{ncopies}, N, P, n, p$), $\mathbf{ncopies}$ has now shape (4.11)
4:     $\mathbf{x}, \mathbf{ncopies} \leftarrow$ Rot.-Red($\mathbf{x}, \mathbf{ncopies}, N, P, n, p$), $\mathbf{ncopies}$ has now shape (4.11) and (3.2)
5: end if
6: $\mathbf{x} \leftarrow$ S-R($\mathbf{x}, \mathbf{ncopies}, n$)
```

4.3.7 Stochastic Volatility

As we can see in Figure 4.4, the PF using Algorithm 28 is up to five times faster than the same using N-R or B-R. The maximum recorded worst-case speed-up is roughly 110 for $P = 256$ cores. The gap between the novel approach and N-R/B-R also increases with DOP as expected after the results in the previous section.

Figure 4.5 shows a full profiling of the three compared PFs for $N = 2^{24}$ and all values of P. We can observe that the $O(\log_2 N)$ redistribute only becomes the bottleneck over

IS for $P = 128$, while B-R and N-R emerge for a relatively low DOP, $P = 2$ for B-R and $P = 16$ for N-R.

These results are remarkable but the performance can be improved even further. The next section shows how it is possible to slightly change the mathematics of Algorithm 28, such that one of the three phases can be entirely skipped with very little overhead, saving 33% of messages.

4.4 Rotational Nearly Sort and Split Redistribute

The previous section has presented a $O(\log_2 N)$ fully-balanced redistribute for DMAs. This novel algorithm is divided into three phases: the first one, Rotational Nearly Sort, rotates the particles to the left by using an LSB-to-MSB decision making policy; the other two phases, Rotational Scatter and Rotational Redistribute, rotate the particle to the right by using the opposite policy to Rotational Nearly Sort. Therefore, these final two phases share the same rotation strategy. In this section, it is proved how to merge Rotational Scatter and Rotational Redistribute into a single phase that requires the same components of Rotational Scatter and also scales as $O(\log_2 N)$. This means that Rotational Redistribute can be entirely removed, which saves one-third of the total messages. The reader is now referred to Figure 4.6, which describes graphically the updated routine by an example with $N = 8$ particles and $P = 4$ cores.

4.4.1 Rotational Split

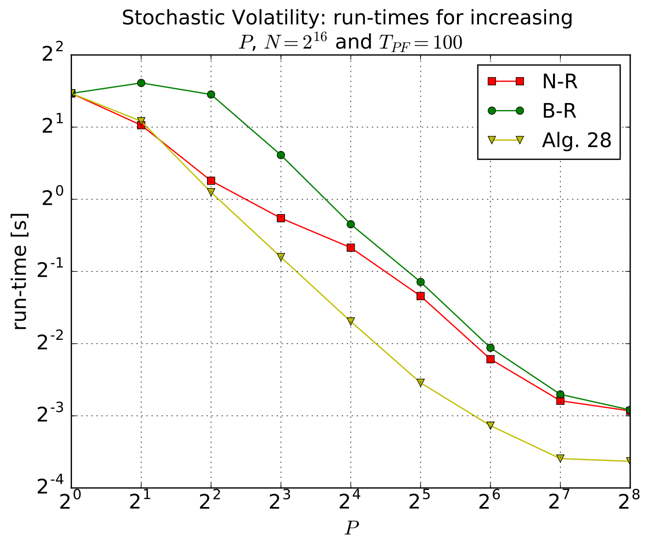
After Rotational Scatter, **ncopies** has shape (4.11) but not (3.2). That is because for each $\mathbf{x}^i$ that must be copied more than once, all its copies are rotated by the same minimum number of positions. However, some of these copies could be split and rotated further to also fill in the gaps that a strategy for (4.11) alone would create.

Therefore, this algorithm also considers the maximum number of rotations that any copy of $\mathbf{x}^i$ has to perform, without causing collisions. Since (4.11) aims to creating $\mathbf{ncopies}^i - 1$ gaps for each i such that $\mathbf{ncopies}^i > 0$, that number is:

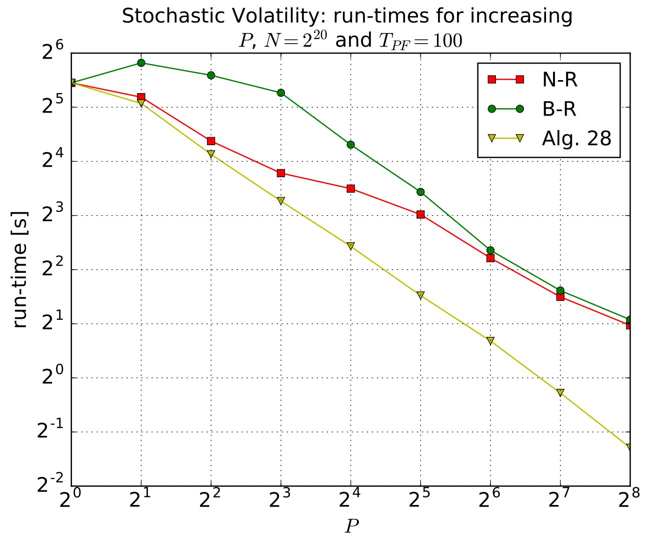
$$\begin{aligned} \mathbf{max\_shifts}^i &= \mathbf{min\_shifts}^i + \mathbf{ncopies}^i - 1 \\ &= \mathbf{csum}^i - i - 1 \end{aligned} \tag{4.16}$$

At each k-th stage of the same binary tree of Rotational Scatter, Rotational Split checks the MSB of both $\mathbf{min\_shifts}^i$ and $\mathbf{max\_shifts}^i$ to infer $\mathbf{copies\_to\_send}^i$, the number of copies of $\mathbf{x}^i$ which must rotate by $N2^{-k}$ positions. For each particle $\mathbf{x}^i$, three possible scenarios may occur:

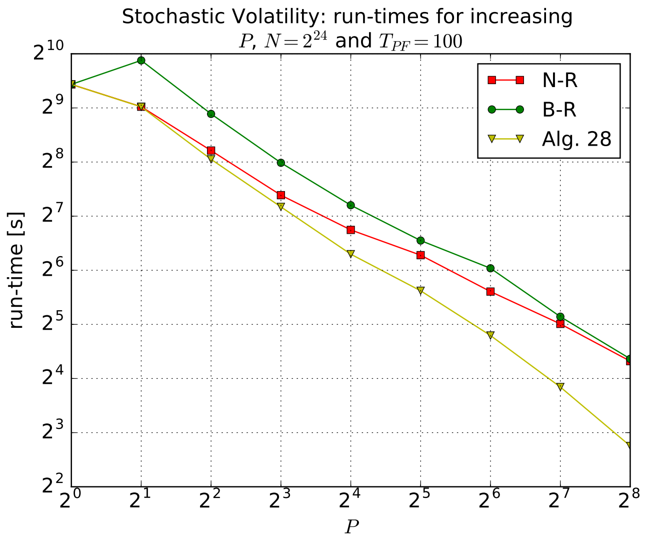
- none of its copies must move;
- all of them must rotate;
- some must split and shift and the others must not move.



(A) $N = 2^{16}$



(B) $N = 2^{20}$



(C) $N = 2^{24}$

FIGURE 4.4: Stochastic Volatility - run-times of PF with $O(\log_2 N)$ redistribute or N-R or B-R for increasing N and P

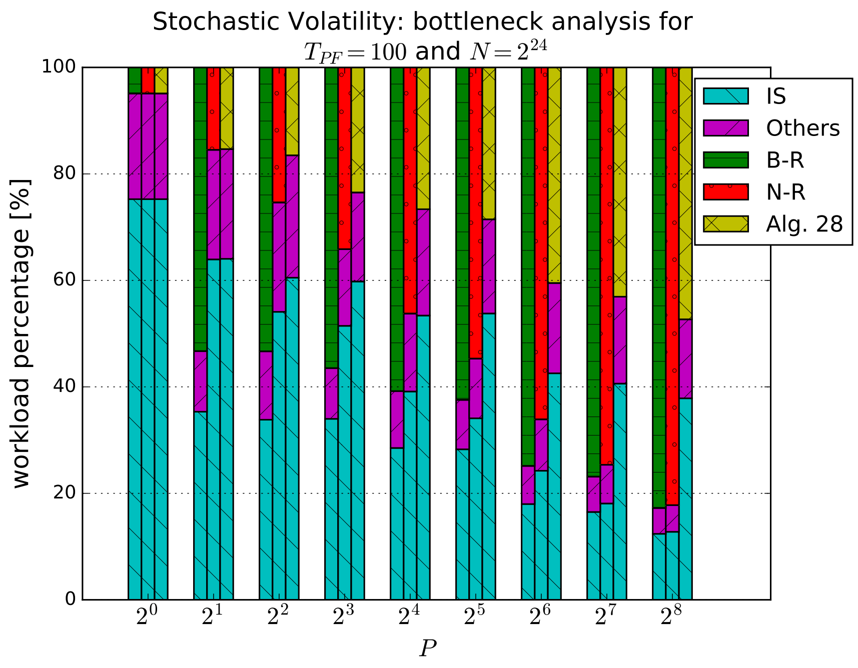


FIGURE 4.5: Stochastic Volatility - bottleneck analysis of PF with Algorithm 28 or N-R or B-R for $N = 2^{24}$

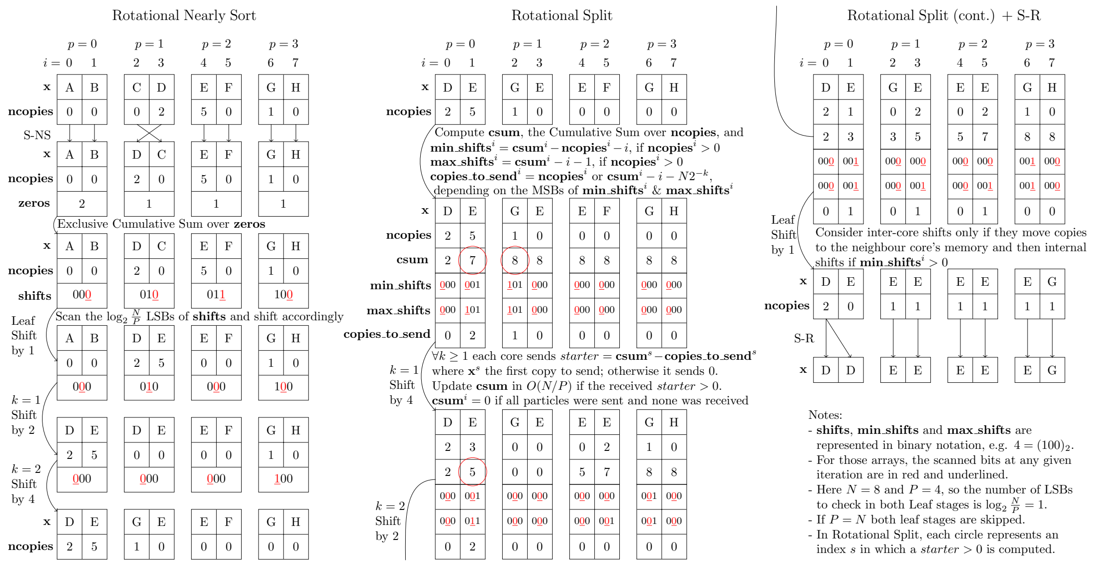

Rotational Nearly Sort
Rotational Split
Rotational Split (cont.) + S-R
$p = 0$
$p = 1$
$p = 2$
$p = 3$
$i = 0$ 1 2 3 4 5 6 7
x
ncopies
S-NS
zeros
Exclusive Cumulative Sum over zeros
shifts
Scan the $\log_2 \frac{N}{P}$ LSBs of shifts and shift accordingly
Leaf Shift by 1
$k = 1$ Shift by 2
$k = 2$ Shift by 4
Compute csum, the Cumulative Sum over ncopies, and
min_shifts^i = csum^i − ncopies^i − i, if ncopies$^i > 0$
max_shifts^i = csum^i − i − 1, if ncopies$^i > 0$
copies_to_send^i = ncopies^i or csum^i − i − $N2^{-k}$,
depending on the MSBs of min_shifts^i & max_shifts^i
csum
min_shifts
max_shifts
copies_to_send
$\forall k \geq 1$ each core sends $starter$ = csum^s − copies_to_send^s
where x^s the first copy to send; otherwise it sends 0.
Update csum in $O(N/P)$ if the received $starter > 0$.
csum$^i = 0$ if all particles were sent and none was received
$k = 1$ Shift by 4
$k = 2$ Shift by 2
Leaf Shift by 1
Consider inter-core shifts only if they move copies to the neighbour core's memory and then internal shifts if min_shifts$^i > 0$
S-R
Notes:
- shifts, min_shifts and max_shifts are represented in binary notation, e.g. $4 = (100)_2$.
- For those arrays, the scanned bits at any given iteration are in red and underlined.
- Here $N = 8$ and $P = 4$, so the number of LSBs to check in both Leaf stages is $\log_2 \frac{N}{P} = 1$.
- If $P = N$ both leaf stages are skipped.
- In Rotational Split, each circle represents an index s in which a $starter > 0$ is computed.

FIGURE 4.6: Rotational Nearly Sort and Split - example for $N = 8$ and $P = 4$

Trivially, the copies will not move if the MSB of **max_shifts**^{i} is 0, which also implies that the MSB of **min_shifts**^{i} is 0, since **min_shifts**$^i \leq$ **max_shifts**^{i} at all stages. If both are equal to 1, the core sends **copies_to_send**^{i} = **ncopies**^{i} copies of $\mathbf{x}^i$. However, if only the MSB of **max_shifts**^{i} is equal to 1, **copies_to_send**^{i} < **ncopies**^{i}. The number of copies to split is equal to how many of them must be placed from position $i + N2^{-k}$ on to achieve perfect workload balance. This is equivalent to computing how many copies make **csum**$^i > i + N2^{-k}$. Therefore, if only the MSB of **max_shifts**^{i} is equal to 1, the core sends

$$\mathbf{copies\_to\_send}^i = \mathbf{csum}^i - i - N2^{-k} \tag{4.17}$$

copies of $\mathbf{x}^i$ and keep the remaining ones still.

If $P < N$, after $\log_2 P$ stages the cores perform once again a leaf stage. Here, the $\log_2 \frac{N}{P}$ LSBs of $\mathbf{min\_shifts}^i$ and $\mathbf{max\_shifts}^i$ are checked at once. In this case, inter-core shifts or splits and shifts are performed only if they send copies to the neighbor. Internal shifts are also considered only if $\mathbf{min\_shifts}^i > 0$, to make enough room to receive particles from the neighbor, since $\mathbf{min\_shifts}^i = \mathbf{max\_shifts}^{i-1} + 1$ (see (4.12) and (4.16)).

In order to ensure logarithmic time complexity, one needs to update all elements in **csum**, **min_shifts** and **max_shifts** in $O(\frac{N}{P})$. This can be done if the cores send *starter* $= \mathbf{csum}^j - \mathbf{copies\_to\_send}^j$, where j is the index of the first particle to send, having $\mathbf{ncopies}^j > 0$. It is once again possible to prove that a particle cannot overwrite or get past another one. This statement will be proven at the end of this section. Hence, each core can safely see the received *starter* as $\sum \mathbf{ncopies}$ for the cores with lower rank, and use it to re-initialise and update **csum** in $O(\frac{N}{P})$ as in Rotational Scatter. Once again, this strategy updates correctly **csum** for at least any index i such that $\mathbf{ncopies}^i > 0$, but those are the only values that have to be correct. Once **csum** is updated, (4.12) and (4.16) are embarrassingly parallel.

It is easy to infer that both $\mathbf{min\_shifts}^i < N - 1$ and $\mathbf{max\_shifts}^i \leq N - 1$, as a particle copy could at most be shifted, or split and shifted from the first to the last position. Therefore, the achieved time complexity of Rotational Split is $O(\log_2 N)$ because the shifts or the split and shifts to perform decrease stage by stage by up to a factor of two. The following theorem proves that a particle can never collide or get past another one.

Theorem 4.6. *Given a nearly-sorted input* **ncopies***, at the k-th iteration of Rotational Split, $\forall k = 1, 2, ..., \log_2 P$, a particle $\mathbf{x}^i$, having* $\mathbf{ncopies}^i \geq 1$ *and rotating to the right by $N2^{-k}$ positions, can never collide with or get past a particle $\mathbf{x}^j$, having* $\mathbf{ncopies}^j \geq 1$ *and $j = i + dist$ with $1 \leq dist \leq N2^{-2k}$.*

Proof of Theorem 4.6. Since Rotational Split uses the same rotation strategy as Rotational Scatter, this theorem can be proved in two possible complementary cases, the same considered in Theorem 4.5:

1. there is one or more zeros between i and j;

2. there are no zeros between i and j.

Case 1. This case is proven the same way *Case 1* of Theorem 4.5 is. Here, the only difference is that having one or more zeros between $\mathbf{x}^i$ and $\mathbf{x}^j$ means that these two particles are at least $dist > \mathbf{max\_shifts}^i \geq N2^{-k}$ locations apart, due previous rotations of $\mathbf{x}^j$ by a previous MSB. Further details are omitted for brevity.

Case 2. In this case, all particles between i and j are still nearly-sorted. Therefore, $\mathbf{x}^i$ would collide with (when $dist = N2^{-k}$) or get past $\mathbf{x}^j$ (when $dist < N2^{-k}$) if

$$\Big(\big(\mathbf{max\_shifts}^i \& N2^{-k}\big) > 0\Big) \wedge \Big(\big(\mathbf{min\_shifts}^j \& N2^{-k}\big) = 0\Big) \tag{4.18}$$

is true, which corresponds to checking whether the MSB of $\mathbf{max\_shifts}^i$ is 1 (which also includes those cases where the MSB of $\mathbf{min\_shifts}^i$ is 1) and the MSB of $\mathbf{min\_shifts}^j$ is 0. In other words, (4.18) can be simplified to checking if $\mathbf{max\_shifts}^i > \mathbf{min\_shifts}^j$. However, for a pair of particles $\mathbf{x}^i$ and $\mathbf{x}^j$ within a nearly-sorted group of particles, that is impossible because:

$$\begin{aligned}
\mathbf{max\_shifts}^i &= \mathbf{csum}^i - i - 1 \\
&= \mathbf{csum}^j - \sum\nolimits_{z=i+1}^{j} \mathbf{ncopies}^z - j + dist - 1 \\
&= \mathbf{csum}^j - \mathbf{ncopies}^j - j - \Big(\sum\nolimits_{z=i+1}^{j-1} \mathbf{ncopies}^z - dist + 1\Big) \\
&= \mathbf{min\_shifts}^j - \Big(\sum\nolimits_{z=i+1}^{j-1} \mathbf{ncopies}^z - dist + 1\Big) \leq \mathbf{min\_shifts}^j
\end{aligned}$$

since $\mathbf{csum}^j = \sum_{z=0}^{j} \mathbf{ncopies}^z$, $dist = j - i$ and (in this case) $\sum_{z=i+1}^{j-1} \mathbf{ncopies}^z \geq j - i - 1$. □

4.4.2 Rotational Nearly Sort and Split

After Rotational Split, S-R can be invoked locally and will finish in $O(\frac{N}{P})$ iterations because the particles are now equally distributed across the MPI ranks.

The overall redistribute will then perform in sequence Rotational Nearly Sort and Rotational Split only if $P > 1$, before calling S-R. The chosen name for this novel algorithm is Rotational Nearly Sort and Split (RoSS) redistribute. We can infer that the achieved time complexity is again $O(N)$ for $P = 1$, $O(\log_2 N)$ for $P = N$ cores and (4.15) for any $1 \leq P \leq N$, but the constant time is about 33% smaller than Algorithm 28 as one phase is now entirely skipped.

Now all tasks in the PF take either $O(1)$ or $O(\log_2 N)$ computations, which brings down the overall time complexity of PFs to $O(\log_2 N)$. This is because, even if we had a $O(1)$ fully-balanced redistribute, the time complexity of PFs would still be bound to $O(\log_2 N)$ because `Normalise`, `ESS`, `MVR` and `Estimate` require reduction or Cumulative Sum (see Table 4.1). The same can be said for any SMC method since the only difference between Algorithms 1, 3 and 4 is that SMC samplers also need `Recycling` which requires reduction.

4.4.3 Algorithmic Implementation

Algorithm 29 describes a pseudo-code for a generic core p running Rotational Split on a DMA. Once again, the for loops, the particles and all arrays related to them are indexed using local indexes, while the description in Section 4.4.1 uses global indexes. Therefore,

TABLE 4.1: Time complexity of each task of SMC methods on DMAs.

Task name (parallelisation strategy)	**Sequential Time Complexity**	**Parallel Time Complexity**
`Initialise` (Embarrassingly parallel)	$O(N)$	$O(1)$
`IS` (Embarrassingly parallel)	$O(N)$	$O(1)$
`Normalise` (Reduction)	$O(N)$	$O(\log_2 N)$
`ESS` (Reduction)	$O(N)$	$O(\log_2 N)$
`MVR` (Cumulative Sum)	$O(N)$	$O(\log_2 N)$
`Redistribute` (RoSS)	$O(N)$	$O(\log_2 N)$
`Reset` (Embarrassingly parallel)	$O(N)$	$O(1)$
`Estimate` (Reduction)	$O(N)$	$O(\log_2 N)$
`Recycling` (Reduction)	$O(N)$	$O(\log_2 N)$

certain terms in some equations, e.g. (4.16) and (4.17), have to be rescaled by $p \times n$ to convert local indexes into global indexes.

Algorithm 30 illustrates RoSS redistribute which invokes Rotational Nearly Sort (see 25) and Rotational Split if $P > 1$, before calling S-R.

4.4.4 Possible non-deterministic optimisations

A trivial idea that can save messages consist of checking the maximum MSB for all **shifts**^{i} and all **max_shifts**^{i} at the beginning of Rotational Nearly Sort and Rotational Split respectively. This will allow us to finish early the execution of either of the two steps and avoid sending unnecessary messages. For example, if **ncopies** happens to be nearly-sorted already, all bits in **shifts** will obviously be 0, meaning that no messages should be sent and Rotational Nearly Sort could be skipped entirely. However, this practice could be faster but might also be slower as it is highly non-deterministic; hence it is strongly not recommended for real-time applications.

4.4.5 Numerical Results

This section shows the numerical results of the improvements on redistribution first and then for two exemplary models, one being again the stochastic volatility model, and the second being the non-linear model in [52, 53], for which $N > 2^{16}$ particles were proven necessary to meet accuracy constraints and up to $N = 2^{24}$ were used. The testing strategy is the same as Section 4.3.5.

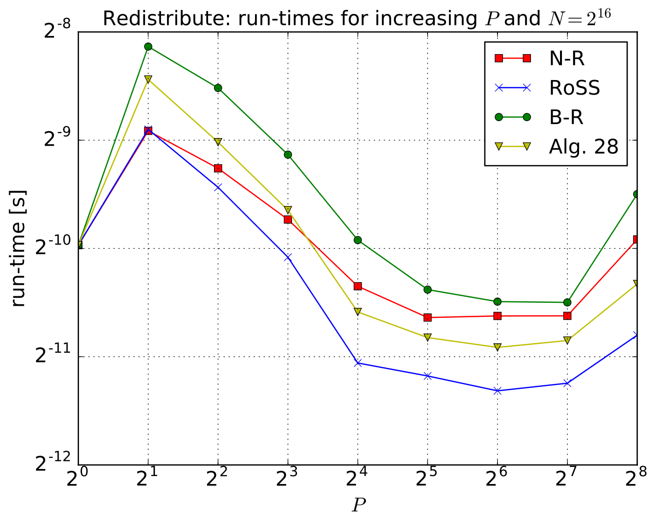
4.4.5.1 Redistribute

Figure 4.7 compares RoSS with Algorithm 28, N-R and B-R. We can see that RoSS optimises the performance of the previous $O(\log_2 N)$ redistribute by a solid 25% factor for any $P > 1$, as it saves one-third of the total messages. This discrepancy is because Rotational Redistribute in Algorithm 28 does not require Cumulative Sum or its updating routine. However, the most interesting results are those in comparison with B-R

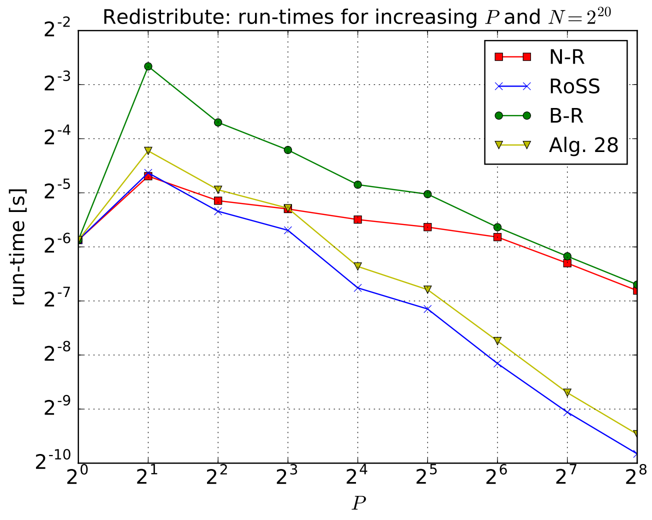
Algorithm 29 Rotational Split

```
Input: $\mathbf{x}$, $\mathbf{ncopies}$, $N$, $P$, $n = \frac{N}{P}$, $p$
Output: $\mathbf{x}$, $\mathbf{ncopies}$
1: $\mathbf{csum} \leftarrow \texttt{Cumulative\_Sum}(N, P, \mathbf{ncopies})$
2: $\mathbf{min\_shifts}^j \leftarrow \mathbf{csum}^j - \mathbf{ncopies}^j - j - np$, $\forall j < n$ if $\mathbf{ncopies}^j > 0$
3: $\mathbf{max\_shifts}^j \leftarrow \mathbf{csum}^j - j - 1 - np$, $\forall j < n$ if $\mathbf{ncopies}^j > 0$
4: for $k \leftarrow 1$; $k \leq \log_2 P$; $k \leftarrow k + 1$ do binary tree
5:     $partner \leftarrow (p + \frac{P}{2^k})\ \&(P - 1)$
6:     for $j \leftarrow 0$; $j < n$; $j \leftarrow j + 1$ do
7:         if $\mathbf{max\_shifts}^j\ \&\ N2^{-k} > 0$ then
8:             if $\mathbf{min\_shifts}^j\ \&\ N2^{-k} > 0$ then
9:                 $\mathbf{copies\_to\_send}^j \leftarrow \mathbf{ncopies}^j$, $\mathbf{ncopies}^j \leftarrow 0$
10:            else
11:                $\mathbf{copies\_to\_send}^j \leftarrow (\mathbf{csum}^j - j - N2^{-k} - pn)$
12:                $\mathbf{ncopies}^j \leftarrow \mathbf{ncopies}^j - \mathbf{copies\_to\_send}^j$
13:            end if
14:            $starter \leftarrow \mathbf{csum}^j - \mathbf{copies\_to\_send}^j$ if $j$ is first
15:            Send $\mathbf{x}^j$, $\mathbf{copies\_to\_send}^j$ to $partner$ and send $starter$ also if $j$ is first
16:        else
17:            Send 0s to $partner$ (Message to reject)
18:        end if
19:    end for
20:    Accept or reject the received particles and $starter$, reset $starter$ to 0 if all
           particles are sent and none is accepted
21:    $\mathbf{csum}^0 \leftarrow starter + \mathbf{ncopies}^0$
22:    $\mathbf{csum}^j \leftarrow \mathbf{csum}^{j-1} + \mathbf{ncopies}^j\ \forall j = 1, 2, \ldots, n - 1$
23:    Update $\mathbf{min\_shifts}$ and $\mathbf{max\_shifts}$ as in steps 2 and 3
24: end for
25: if $P < N$ then perform leaf stage of the binary tree
26:    for $j \leftarrow n - 1$; $j \geq 0$; $j \leftarrow j - 1$ do
27:        if $\mathbf{csum}^j > (p + 1)n$ then
28:            $\mathbf{copies\_to\_send}^j \leftarrow \min(\mathbf{csum}^j - (p + 1)n, \mathbf{ncopies}^j)$
29:            $\mathbf{ncopies}^j \leftarrow \mathbf{ncopies}^j - \mathbf{copies\_to\_send}^j$
30:            Send $\mathbf{x}^j$, $\mathbf{copies\_to\_send}^i$ to $partner$
31:        else
32:            Send 0s to $partner$ (Message to reject)
33:        end if
34:        if $\mathbf{min\_shifts}^j > 0$ then
35:            Shift particle to the right by $\mathbf{min\_shifts}^j$
36:        end if
37:    end for
38:    Accept or reject the received particles
39: end if
```

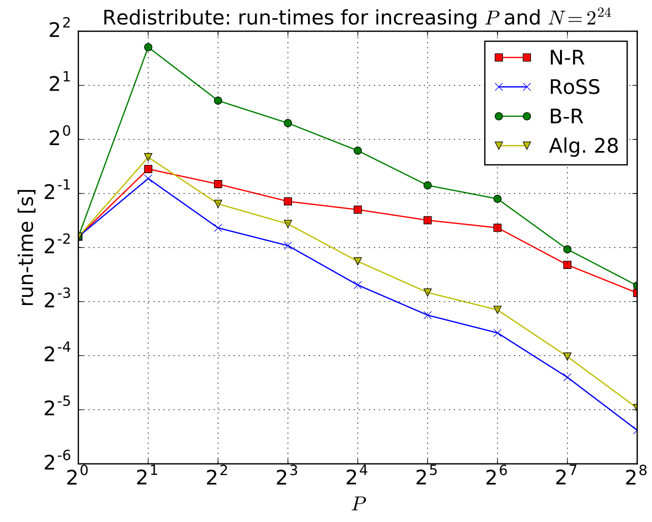
and N-R, both presented in the previous chapter. RoSS redistribute is now comparable with N-R and up to four times as fast as B-R for $P = 2$ cores. Most importantly, as the DOP increases, the speed-up vs B-R and N-R improves consistently, such that RoSS is eight to nine times faster than N-R and B-R for $P = 256$.



(A) $N = 2^{16}$



(B) $N = 2^{20}$



(C) $N = 2^{24}$

FIGURE 4.7: RoSS vs B-R vs N-R - run-times for increasing N and P

Algorithm 30 Rotational Nearly Sort and Split (RoSS)

```
Input: $\mathbf{x}$, ncopies, $N$, $P$, $n = \frac{N}{P}$, $p$
Output: $\mathbf{x}$
1: if $P > 1$ then
2:     $\mathbf{x}, \mathbf{ncopies} \leftarrow$ Rot.-NS($\mathbf{x}, \mathbf{ncopies}, N, P, n, p$), ncopies has now shape (3.8)
3:     $\mathbf{x}, \mathbf{ncopies} \leftarrow$ Rot.-Split($\mathbf{x}, \mathbf{ncopies}, N, P, n, p$), (4.11) and (3.2) now hold
4: end if
5: $\mathbf{x} \leftarrow$ S-R($\mathbf{x}, \mathbf{ncopies}, n$)
```

4.4.5.2 Stochastic Volatility

As we can see in Figure 4.8, the PF using RoSS is superior to any other PF option. Most importantly, it is up four to six times faster than a PF using N-R/B-R. The maximum speed-up is about 125 for $P = 256$ cores. The gap between a PF with RoSS and a PF with N-R/B-R also increases with P as expected after the results in the previous section. It is also remarkable to see that RoSS is the only redistribute option which remains faster than IS for any P here, as shown in Figure 4.10a.

4.4.5.3 Vacuum Arc Remelting

Vacuum Arc Remelting (VAR) is a secondary melting process, used in the final stage of alloy productions. In this stage, a continuous arc strikes between an electrode and a solidifying ingot, making the metal melt off the electrode and then fall onto a melt pool to solidify. Since this process is done in a vacuum, a lot of impurities are removed, resulting in higher quality of the finished product. During the VAR process, it is fundamental to keep track of the liquid pool depth which, however, cannot be measured directly. A parallel PF can then be used in this case. The following dynamic model is for Alloy 718 and is thoroughly explained in [52, 53]. Here, for brevity, only the most important details are highlighted.

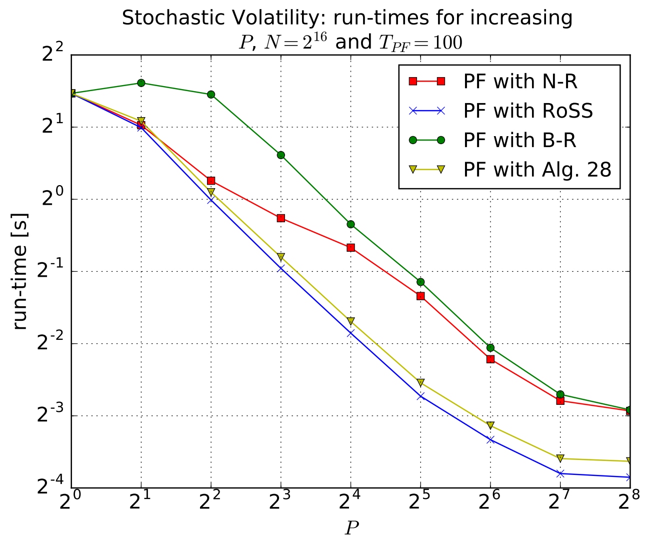
$\mathbf{X}_t$ is nine-dimensional and contains information about the electrode thermal boundary layer Δ, the electrode gap G, the ram position X_{ram}, the electrode mass M_e, the melting efficiency μ, the centerline pool depth S_C, the mid-radius pool depth S_M, the helium pressure p_{he}, and the current I.

$$\Delta_t = \Delta_{t-1} + \left[\frac{\alpha_r C_{\Delta\Delta}}{\Delta_{t-1}} - \frac{C_{\Delta p}}{A_e h_m}\mu_{t-1}(V_c + R_i I_{t-1})I_{t-1}\right] dt + G_{11} dI \quad (4.19a)$$

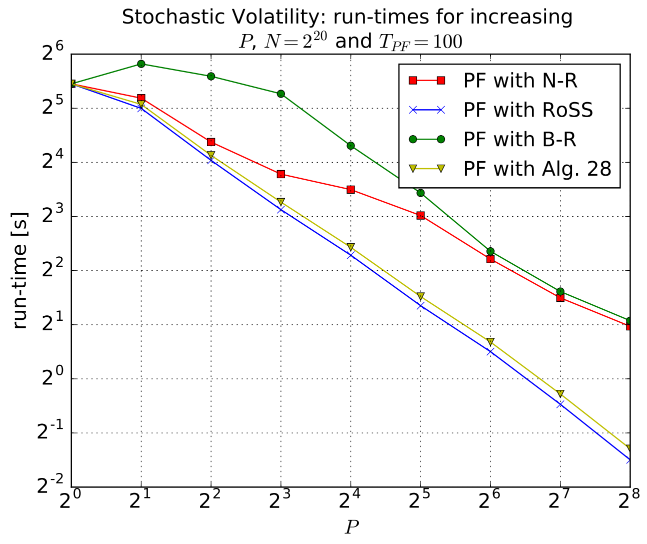
$$G_t = G_{t-1} + \left[-\frac{\alpha_0 \alpha_r C_{s\Delta}}{\Delta_{t-1}} + \frac{\alpha_0 C_{\Delta p}}{A_e h_m}\mu_{t-1}(V_c + R_i I_{t-1})I_{t-1} - V_{ram}\right] dt + G_{21} dI - dV_{ram} \quad (4.19b)$$

$$M_{e_t} = M_{e_{t-1}} + \left[\frac{\rho A_e C_{s\Delta}}{\Delta_{t-1}} + \frac{\rho C_{sp}}{h_m}\mu_{t-1}(V_c + R_i I_{t-1})I_{t-1}\right] dt + G_{41} dI \quad (4.19c)$$

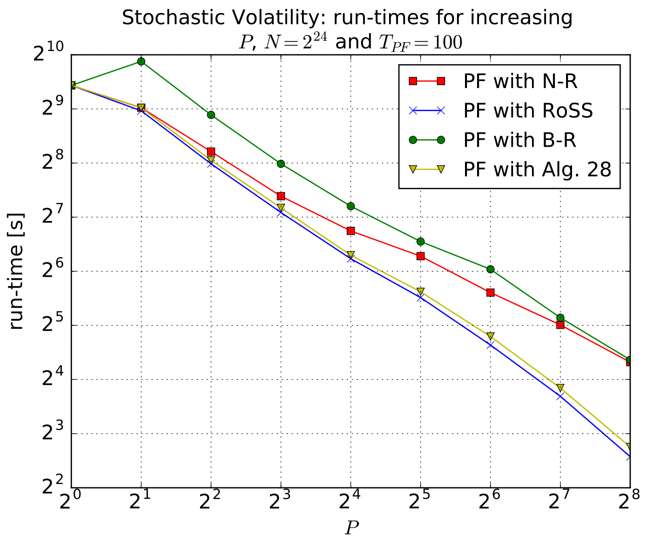
$$X_{ram_t} = X_{ram_{t-1}} + V_{ram} dt + dV_{ram} \quad (4.19d)$$



(A) $N = 2^{16}$



(B) $N = 2^{20}$



(C) $N = 2^{24}$

FIGURE 4.8: Stochastic Volatility - run-times of PF with RoSS or N-R or B-R for increasing N and P

$$\mu_t = \mu_{t-1} + d\mu \tag{4.19e}$$

$$S_{C_t} = S_{C_{t-1}} - \Big[A_C(S_{C_{t-1}} - S_{C_0}) - B_{\Delta C}(\Delta_{t-1} - \Delta_0) -$$
$$B_{iC}(I_{t-1} - I_0) - B_{\mu C}(\mu_{t-1} - \mu_0) - B_{heC}(p_{he_{t-1}} - p_{he_0})\Big]dt - B_{iC}dI \tag{4.19f}$$

$$S_{M_t} = S_{M_{t-1}} - \Big[A_M(S_{M_{t-1}} - S_{M_0}) - B_{\Delta M}(\Delta_{t-1} - \Delta_0) -$$
$$B_{iM}(I_{t-1} - I_0) - B_{\mu M}(\mu_{t-1} - \mu_0) - B_{heM}(p_{he_{t-1}} - p_{he_0})\Big]dt - B_{iM}dI \tag{4.19g}$$

$$p_{he_t} = p_{he_{t-1}} + dhe \tag{4.19h}$$

$$I_t = I_c + (I_c - I_{t-1})e^{-dt/d\tau} + dI \tag{4.19i}$$

where the time interval dt, the melting current I_c and the ram speed V_{ram} are controlled by the user; here we use $dt = 5$s, $I_c = 6000$A and V_{ram} is inferred by setting to 0 the expression between squared brackets in (4.19b). The process noise terms dI, dV_{ram}, $d\mu$ and dp_{he} are Gaussian with 0 mean and covariances $(\sigma_I dt)^2$, $(\sigma_{V_{ram}} dt)^2$, $(\sigma_\mu \mu_0)^2 dt$, $(\sigma_{p_{he}} p_{he_0})^2 dt$ respectively.

$$\mathbf{Y}_t = [G_t, X_{ram_t}, M_{e_t}, S_{C_t}, S_{M_t}, p_{he_t}, I_t, V_c + R_i I_t] + \mathcal{N}(\mathbf{0}, \mathbf{R}) \tag{4.20}$$

where $\mathbf{R} = diag(\sigma^2_G, \sigma^2_X, \sigma^2_{M_e}, \sigma^2_C, \sigma^2_M, \sigma^2_{he_m}, \sigma^2_{I_m}, \sigma^2_{V_m})$.

$$\mathbf{X}_0 \sim \mathcal{N}([\Delta_0, G_0, X_{ram_0}, \mu_0, M_{e_0}, S_{C_0}, S_{M_0}, p_{he_0}, I_0], \mathbf{Q}) \tag{4.21}$$

where $\mathbf{Q} = diag(\sigma^2_\Delta, \sigma^2_G, \sigma^2_X, \sigma^2_{LC}, \sigma^2_\mu dt, \sigma^2_{he} dt, \sigma^2_C, \sigma^2_M, \sigma^2_{I_m})$. Tables 4.2, 4.3 and 4.4 provide all constants and σ terms in (4.19), (4.20), and (4.21). The dynamics is again chosen as proposal.

Symbol	Value	Symbol	Value	Symbol	Value
σ_{he_m}	0.01 Torr	σ_{I_m}	15 A	σ_M	1 cm
σ_X	0.005 cm	σ_G	0.2 cm	σ_C	1 cm
$\sigma_{V_{ram}}$	5e−4 cm/s	σ_{LC}	0.2 kg	σ_Δ	5 cm
σ_μ	$0.001\mu_0$	σ_{V_m}	0.1 V		
σ_{he}	$0.001p_{he_0}$	σ_I	20 A		

Table 4.2: Standard deviations for noise terms.

As we can see in Figure 4.10b, IS now takes up a larger percentage of the total run-time for low P than it does in the previous example. This is because $\mathbf{X}_t$ and $\mathbf{Y}_t$ in (2.12) and (2.13) are more computationally intensive and about $M = 9$ and $M_y = 8$ times bigger, while only redistribution and `Estimate` have increased in dimensionality, among the other tasks. We can indeed observe that IS is the bottleneck over any redistribute for

Symbol	**Value**	**Symbol**	**Value**
$C_{\Delta\Delta}$	40	B_{iM}	3.2e−6 cm/s A
$C_{\Delta p}$	3.8	$B_{\mu C}$	6.9e−4 cm/s
$C_{s\Delta}$	6.7	$B_{\mu M}$	2.7e−4 cm/s
C_{sp}	1.3	B_{heC}	−8.1e−4 cm/s Torr
A_C	1.9e−3 1/s	B_{heM}	−6.5e−4 cm/s Torr
A_M	1.4e−3 1/s	G_{11}	−5.6e−6 cm/A
$B_{\Delta C}$	2.6e−5 1/s	G_{21}	5.4e−7 cm/A
$B_{\Delta M}$	−1.3e−4 1/s	G_{41}	−2.2e−2 g/A
B_{iC}	6.6e−6 cm/s A	$d\tau$	1 s

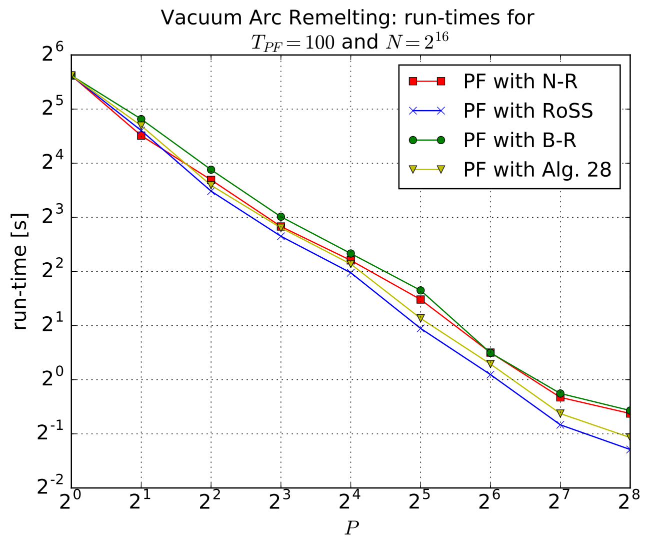
TABLE 4.3: Parameters of Alloy 718.

Name (symbol)	**Value**
Nominal electrode thermal boundary layer (Δ_0)	80 cm
Nominal electrode gap (G_0)	0.9 cm
Nominal ram position (X_{ram_0})	0.7 cm
Nominal electrode mass (M_{e_0})	4800 kg
Nominal melting efficiency (μ_0)	0.44
Nominal centerline pool depth (S_{C_0})	15.7 cm
Nominal mid-radius pool depth (S_{M_0})	13.2 cm
Nominal helium pressure (p_{he_0})	3.0 Torr
Nominal current (I_0)	6000 A
Electrode cross section (A_e)	1460 cm^2
Area fill ratio (a)	0.28
Density (ρ)	8.192 g/cm^3
Cathode voltage fall (V_c)	21.2 V
Electric resistance (R_i)	4.37e−4 Ω
Thermal diffusivity at 300 K (α_r)	2.4e−2 cm^2/s
Thermal diffusivity at 1623 K (α_m)	6.0e−2 cm^2/s
Volume-specific enthalpy at 1623 K (h_m)	5.4e3 J/cm^3
Volume-specific enthalpy at 1673 K (h_s)	8.1e3 J/cm^3

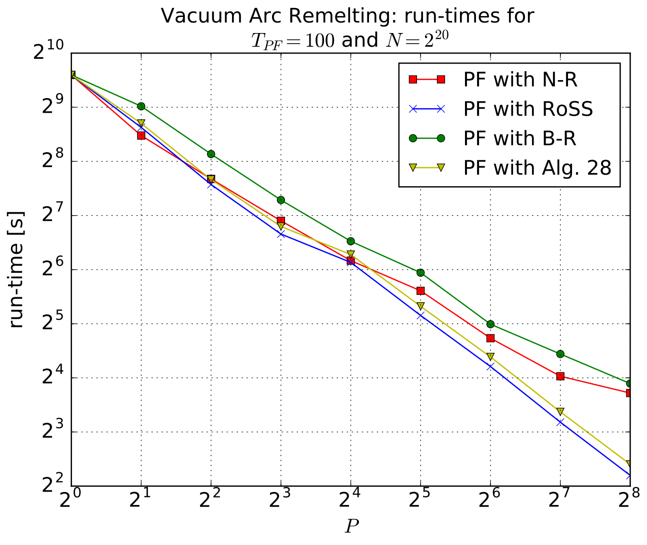
TABLE 4.4: Nominal values, furnace and Alloy 718 properties.

$P = 2$. However, as P increases, both B-R and N-R eventually emerge as the bottleneck for $8 \leq P \leq 32$. On the other hand, it is again interesting to see that RoSS still is faster than IS for any $P \leq 256$. In Figure 4.9, we can observe that a PF with RoSS redistribute outperforms any other PF option, especially the one with N-R/B-R by up to a factor of three, which is once again increasing with P. The maximum recorded speed-up is 160 for a PF with RoSS and about 50 for a PF with N-R/B-R, hence both higher than in the previous example.

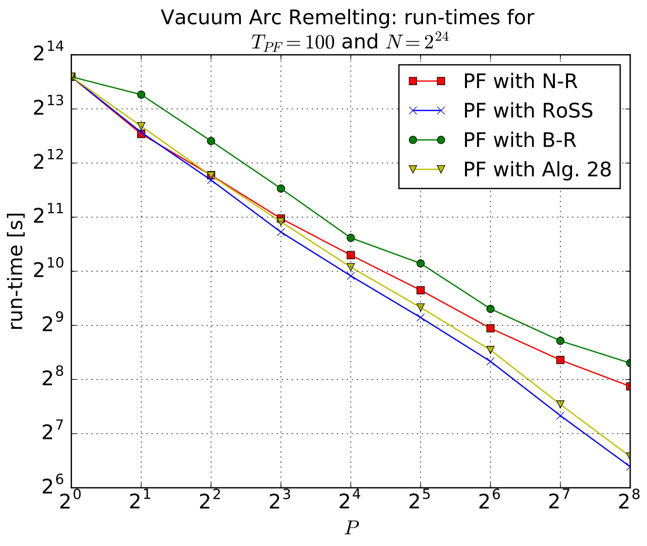
These results underline well the importance of having a fast, scalable redistribution. Since modern dynamic or static models may be very detailed and complex (e.g. requiring some sophisticated numerical integrator) the IS step also becomes highly computationally intensive. Therefore, a fast redistribute allows PFs to maintain a near-linear speed-up for larger P, which is desirable in theory but hard to achieve in practice.



(A) $N = 2^{16}$

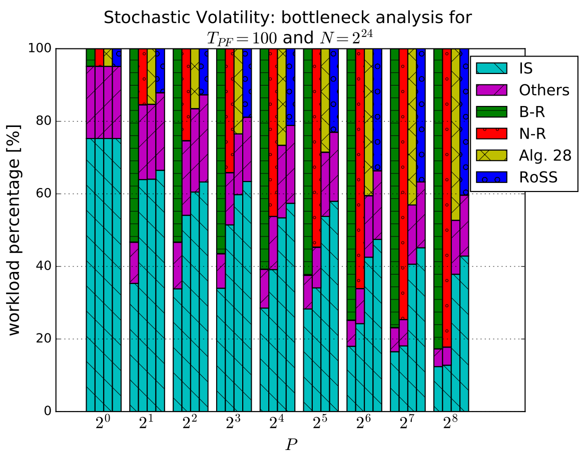


(B) $N = 2^{20}$

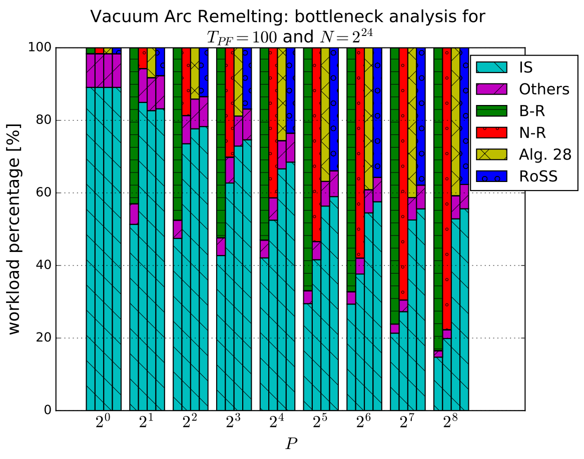


(C) $N = 2^{24}$

FIGURE 4.9: Vacuum Arc Remelting - run-times of PF with RoSS or N-R or B-R for increasing N and P



(A) Stochastic Volatility



(B) Vacuum Arc Remelting

FIGURE 4.10: VAR and Stochastic Volatility with RoSS - bottleneck analysis of PF with RoSS, Algorithm 28, N-R or B-R for $N = 2^{24}$ and increasing P

4.5 Conclusions

This chapter has presented RoSS redistribute, a novel fully-balanced redistribution for SMC methods on distributed memory environments. This algorithm has been implemented on MPI. The baselines for comparison are B-R and N-R, two similar MPI fully-balanced redistribute algorithms which both achieve $O((\log_2 N)^2)$ time complexity and whose implementation is described in Chapter 3. However, in this chapter it has been proved that RoSS redistribute achieves exactly $O(\log_2 N)$ time complexity, the lower bound for SMC methods.

The results in Section 4.4.5 show that the RoSS redistribute is almost an order of magnitude faster than B-R and N-R for up to $P = 256$ cores. Similar results are observed on two exemplary PFs too. For the same level of parallelism, a PF using RoSS redistribute is up to six times faster than a PF using B-R/N-R and provides a maximum speed-up of 160 vs a single-core PF. It is also interesting to highlight that, under the same testing conditions, RoSS is the only option for redistribute which is still faster than IS in both models.

The improvements are encouraging but several advances can still be made. As done in this chapter, the focus should be on keeping investigating solutions to reduce the number of messages between the cores. One key observation is that the current implementation is MPI-everywhere, which means that the memory is distributed not only across inter-node processors, but also between cores which are embedded into the same compute node. In reality, this is not the case. Such an architectural arrangement can be exploited by using the shared-memory parallelism within nodes and distributed memory parallelism across nodes. Mixing OpenMP, one of the most common programming models for SMAs, with MPI is indeed a routine practice in the HPC domain.

The following chapter shows how to derive an MPI+OpenMP version of RoSS from the implementation described in this chapter. In order not to lose the advances made so far, one needs to ensure every single-core component in RoSS achieves either $O(1)$ or $O(\log_2 N)$ time complexity on OpenMP. Therefore, a novel $O(\log_2 N)$ shared-memory implementation of redistribute is also developed in the next chapter.

Chapter 5

A Fast Parallel Particle Filter on Hybrid Memory Architectures

5.1 Introduction

The previous chapter presents a novel fully-balanced redistribute algorithm for SMC methods on DMAs which achieves $O(\log_2 N)$ time complexity. Although DMAs are frequently used, especially in HPC contexts, there are other memory environments that can be exploited. Share Memory Architectures (SMAs), such as shared-memory CPUs and GPUs, represent a possible alternative. However, on SMAs, as well as on DMAs, an efficient parallelisation of SMC methods depends once again on how effectively the redistribution step can be parallelised.

On DMAs, fully-balanced redistribute parallelisations use a load-balancing routine before redistributing. In the previous chapters, the reader can see that these load-balancing routines typically require a sorting or nearly-sorting phase. On SMAs, however, it has been proven that sort is optional as it can be substituted by N Binary Searches. This idea was implemented in [52], optimised in [53] and applied to different resampling schemes in [62]. The time complexity is $O(\frac{N}{T}\log_2 N)$ for T parallel shared-memory cores. Since N could be large, due to demanding accuracy constraints [53], this redistribute strategy cannot be effectively parallelised due to the inherently limited number of cores of modern SMAs.

This chapter has two goals. The first is to propose a novel SMA redistribute parallelisation on OpenMP which takes $O(\frac{N}{T}+\log_2 N)$ steps and fully-exploits the computational power of SMAs. In HPC applications, DMAs and SMAs are often combined to optimise the performance, whenever a higher DOP does not offer further scalability in DMA-only contexts. Therefore, the second goal is to investigate the possible advantages of using a hybrid DMA-SMA approach for a parallel SMC method, by combining the MPI algorithms developed in Chapters 3 and 4 with the single-node OpenMP algorithm presented in this chapter. In doing so, the next two sections of this chapter are dedicated to each goal. Every time, results for both redistribute and an exemplary PF are provided.

Section 5.4 draws the final conclusions of this chapter and gives suggestions for future improvements. The reader is recommended to consult Appendix A.2 for details about SMAs and OpenMP and Appendix A.3 for Hybrid Memory Architectures (HMAs).

The results in Section 5.2 have been published in [91].

5.2 An $O(\log_2 N)$ OpenMP Particle Filter

This section illustrates how to parallelise each task of PFs on OpenMP, including a novel implementation of redistribute. It also gives the results for each task as a standalone and for the stochastic volatility model which easily generates the worst-case scenario.

The results in this section are provided both on mainstream CPUs and on GPU, to resemble the narrative in [91]. Since two different types of SMAs are employed, in this section only the run-times may be plotted as function of N, instead of the DOP only as in the previous chapter. In the following sections, the pseudo-codes are illustrated for shared-memory CPUs only for brevity reasons, but an OpenMP 4.5 algorithm can target GPU by simply adding `target teams distribute map` clauses to the existing `#pragma omp parallel for` directives. In this section, there have been made lots of optimisations on several tasks that are possible on OpenMP and are not used elsewhere, e.g. optimisations of the Random Number Generator (RNG) used in (2.12), or the use of SIMD in all algorithms of this chapter whenever possible, and use of vector implementations of math functions, such as log and exp; details are omitted for brevity.

All results are medians of 20 runs for up to $N = 2^{24}$ as in the previous chapters and different numbers of threads T. The CPU used is a 2 Xeon Gold 6138 which has up to 40 cores, provided by each computing node of the cluster in Table 3.2. However only 32 cores are employed as many of these algorithms use the divide-and-conquer paradigm. For the GPU results, the same cluster provides a GPU node mounting an NVIDIA Tesla V100 which has 5120 cores. In this section, Clang 7.0 is used as in [69], as this is one of the most popular compilers that supports GPU offload on NVIDIA graphics cards.

5.2.1 Embarrassingly Parallel

As stated in Chapter 3, `Reset`, `Initialise`, (2.18) and all variants of IS are element-wise operations and hence trivially parallelisable. On OpenMP they can be parallelised by `#pragma omp parallel for` directives. Also, the workload is roughly balanced $\forall i = 0, 1, ..., N-1$. Therefore, a static scheduling clause should be added to the `#pragma` directive.

Here, as in Section 3.3.1, only the OpenMP pseudo-code for `Reset` and IS are illustrated. Figure 5.1 shows the most significant run-times for IS on both CPU and GPU. As we can see, the speed-up for varying DOP increases with N, due to the increased workload per thread. This is true for both the CPU and the GPU. Most importantly, the GPU roughly provides up to a three-fold speed-up vs its equivalent on a 32-thread CPU. This is mostly due to the improvements on the RNG that Curand provides.

Algorithm 31 Importance Sampling (IS)

```
Input: $\mathbf{x}_{t-1}$, $\mathbf{w}_{t-1}$, $\mathbf{Y}_t$, $N$, $T$
Output: $\mathbf{x}_t$, $\mathbf{w}_t$
1: #pragma omp parallel for schedule(static) num_threads($T$)
2: for $i \leftarrow 0$; $i < N$; $i \leftarrow i + 1$ do
3:     $\mathbf{x}_t^i \sim q(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i, \mathbf{Y}_t)$
4:     $\mathbf{w}_t^i \leftarrow \mathbf{w}_{t-1}^i \frac{p(\mathbf{Y}_t|\mathbf{x}_t^i)p(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i)}{q(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i,\mathbf{Y}_t)}$
5: end for
```

Algorithm 32 Reset

```
Input: $\mathbf{w}$, $N$, $T$
Output: $\mathbf{w}$
1: #pragma omp parallel for schedule(static) num_threads($T$)
2: for $i \leftarrow 0$; $i < N$; $i \leftarrow i + 1$ do
3:     $\mathbf{w}^i \leftarrow \frac{1}{N}$
4: end for
```

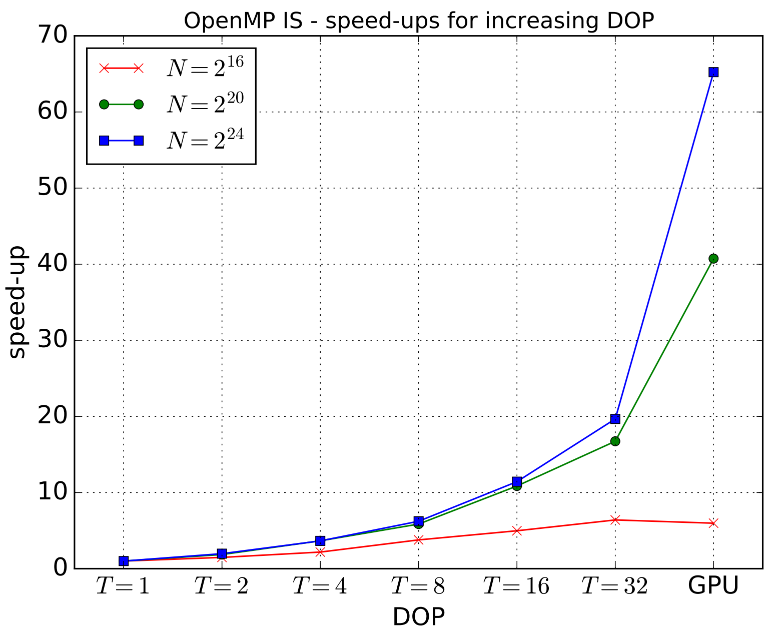


FIGURE 5.1: OpenMP IS - speed-ups for increasing N and DOP

5.2.2 Reduction

As mentioned in Chapter 3, the sum in (2.14), (2.16), (2.19) and (2.15) can be parallelised by using the reduction operation which scales as $O(\frac{N}{T} + \log_2 T)$ on SMAs as well. On OpenMP, reduction can be used by adding to the `#pragma` directives a `reduction` clause applied to the variable to reduce.

Algorithms 33, 34 and 35 describe OpenMP pseudo-codes for `Normalise`, `ESS` and `Mean`. The OpenMP implementation of (2.15) is omitted for brevity as it is almost identical to `Mean`.

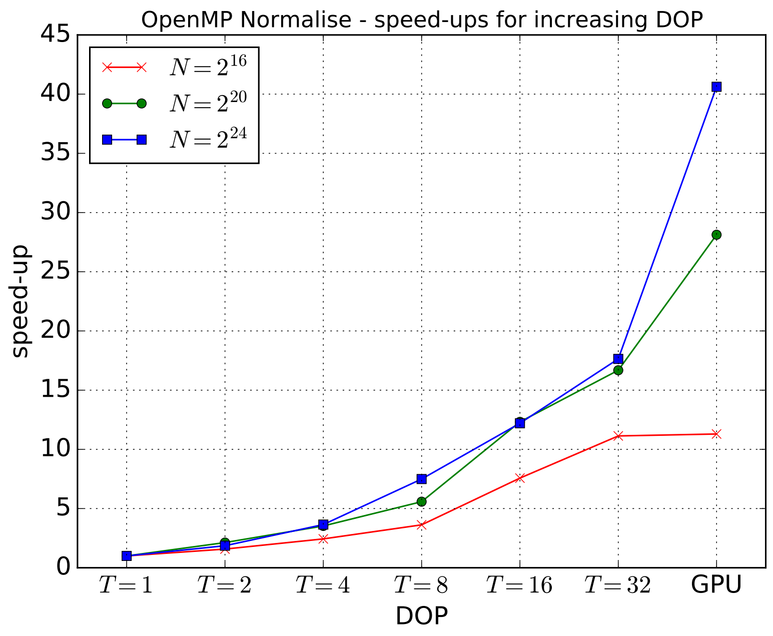


FIGURE 5.2: OpenMP Normalise - speed-ups for increasing N and DOP

For brevity, only the results for `Normalise` are provided in Figure 5.2. Here, as in the previous section, we can see that the maximum speed-up for $T = 32$ along with the speed-up provided by the GPU both increase with N.

Algorithm 33 Normalise

```
Input: $\mathbf{w}$, $N$, $T$
Output: $\tilde{\mathbf{w}}$
1: $local\_sum \leftarrow 0$
2: #pragma omp parallel for reduction(+:$local\_sum$) num_threads($T$)
3: for $i \leftarrow 0$; $i < n$; $i \leftarrow i + 1$ do
4:     $local\_sum \leftarrow local\_sum + \mathbf{w}^i$
5: end for
6: #pragma omp parallel for schedule(static) num_threads($T$)
7: for $i \leftarrow 0$; $i < N$; $i \leftarrow i + 1$ do
8:     $\tilde{\mathbf{w}}^i \leftarrow \frac{\mathbf{w}^i}{sum}$
9: end for
```

5.2.3 Cumulative Sum

As previously mentioned, the CDF of the weights requires Cumulative Sum which also achieves $O(\frac{N}{T} + \log_2 T)$ time complexity on SMAs [60].

Algorithms 36 and 37 illustrate OpenMP pseudo-codes for parallel Cumulative Sum and for MVR respectively.

Figures 5.3 and 5.4 show the most significant run-times on shared memory CPUs and GPU for increasing DOP and N. For these tasks also, we can see that the speed-up for $T = 32$ increases with N. The same can be said for the improvements that the GPU provides vs a 32-core CPU.

Algorithm 34 Effective Sample Size (ESS)

Input: $\tilde{\mathbf{w}}$, N, T,

Output: N_{eff}

```
1: $local\_sum \leftarrow 0$
2: #pragma omp parallel for reduction(+:$local\_sum$) num_threads($T$)
3: for $i \leftarrow 0$; $i < N$; $i \leftarrow i + 1$ do
4:     $local\_sum \leftarrow local\_sum + (\tilde{\mathbf{w}}^i)^2$
5: end for
6: $N_{eff} \leftarrow \frac{1}{local\_sum}$
```

Algorithm 35 Mean

Input: $\mathbf{x}$, N, T, M

Output: ξ

```
1: for $j \leftarrow 0$; $i < M$; $j \leftarrow j + 1$ do
2:     $\mathbf{local\_sum}^j \leftarrow 0$, **local_sum** is in bold here because $\mathbf{x}^i \in \mathbb{R}^M$
3:     #pragma omp parallel for reduction(+:$\mathbf{local\_sum}^j$) num_threads($T$)
4:     for $i \leftarrow 0$; $i < N$; $i \leftarrow i + 1$ do
5:         $\mathbf{local\_sum}^j \leftarrow \mathbf{local\_sum}^j + \mathbf{x}^{i,j}$
6:     end for
7: end for
8: $\xi \leftarrow \frac{\mathbf{local\_sum}}{N}$
```

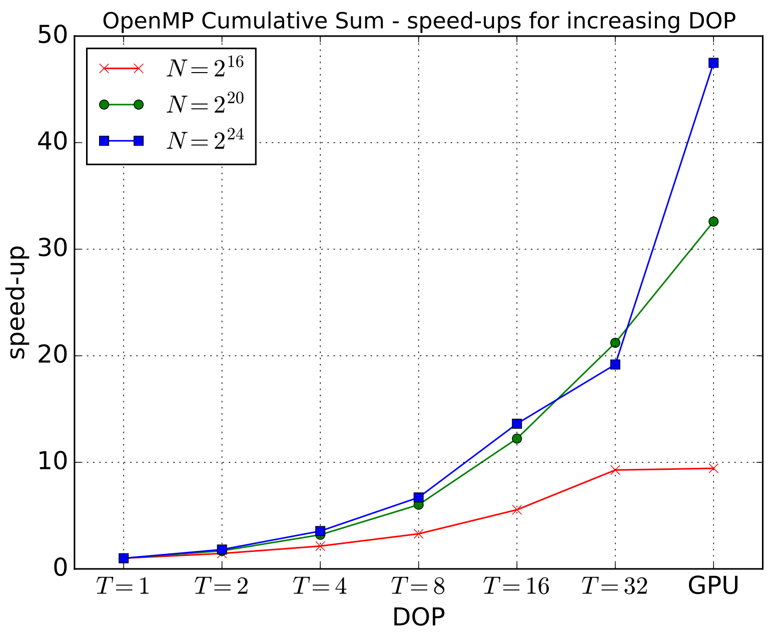


FIGURE 5.3: OpenMP Normalise - speed-ups for increasing N and DOP

5.2.4 A Novel $O(\log_2 N)$ Redistribute on Shared Memory Systems

In Chapters 3 and 4, it has been stated that S-R has a low constant time (it only consists of N memory writes) but is impossible to parallelise in an element-wise fashion.

Algorithm 36 OpenMP Cumulative Sum

```
Input: array, $N$, $T$, $form$
Output: csum
#pragma omp parallel num_threads($T$){
    $id \leftarrow$omp_get_thread_num(), $base \leftarrow id \times \frac{N}{T}$
    $\mathbf{csum}^{base} \leftarrow \mathbf{array}^{base}$
    for $i \leftarrow base + 1$; $i < base + n$; $i \leftarrow i + 1$ do
        $\mathbf{csum}^{i} \leftarrow \mathbf{csum}^{i-1} + \mathbf{array}^{i}$
    end for
    $\mathbf{sum}^{id} \leftarrow \mathbf{csum}^{base+\frac{N}{T}-1}$
    $\mathbf{coreSum}^{id} \leftarrow \mathbf{csum}^{base+\frac{N}{T}-1}$
}
for $dist \leftarrow 1$; $dist < T$; $dist \leftarrow dist \times 2$ do
    #pragma omp parallel num_threads($T$){
        $id \leftarrow$omp_get_thread_num(), $partner \leftarrow id \oplus dist$
        $\mathbf{sum}^{id} \leftarrow \mathbf{sum}^{id} + \mathbf{sum}^{partner}$
        if $id > partner$ then
            $\mathbf{coreSum}^{id} \leftarrow \mathbf{coreSum}^{id} + \mathbf{sum}^{partner}$
        end if
    }
end for
#pragma omp parallel num_threads($T$){
    $id \leftarrow$omp_get_thread_num(), $base \leftarrow id \times \frac{N}{T}$
    if $form ==$ Inclusive then
        $\mathbf{csum}^{base} \leftarrow \mathbf{coreSum}^{id} - \mathbf{csum}^{base+\frac{N}{T}-1} + \mathbf{array}^{base}$
    else
        $\mathbf{csum}^{base} \leftarrow \mathbf{coreSum}^{id} - \mathbf{csum}^{base+\frac{N}{T}-1}$
    end if
    for $i \leftarrow base + 1$; $i < base + n$; $i \leftarrow i + 1$ do
        $\mathbf{csum}^{i} \leftarrow \mathbf{csum}^{i-1} + \mathbf{array}^{i}$
    end for
}
```

Each $\mathbf{ncopies}^i$ could randomly be equal to any integer between 0 and N and hence, the workload could be highly unbalanced. This is true on SMAs as well.

Multi-PF approaches can be used to bypass the need of parallelising S-R and require little to no changes from a DMA implementation. However, as explained in Section 3.2, this approach has accuracy, scalability and applicability issues. A centralised approach can also be used [47], but as said in Section 3.2, scalability is not guaranteed for large DOP. This will also be clear in the results of the next section. An alternative fully-balanced parallel redistribution for SMAs on CPU and GPU can be found in [52, 53, 62][1]. The idea is to make use of $\mathbf{csum} \in \mathbb{Z}^N$, the Cumulative Sum of **ncopies**, to search for

[1]This redistribution parallelisation is applied to MVR, the only resampling scheme considered in this thesis. However, using other resampling schemes, such as Stratified Resampling, may also lead to alternative $O(\log_2 N)$ parallel resampling methods for SMAs (see [40]). Cross-comparisons between different resampling schemes are out of the scope of this thesis but to be considered for future work.

Algorithm 37 Minimum Variance Resampling (MVR)

```
Input: $\tilde{\mathbf{w}}$, $N$, $T$
Output: ncopies
1: $\mathbf{cdf}^0 \leftarrow 0$, because $\mathbf{cdf} \in \mathbb{R}^{N+1}$
2: $\mathbf{cdf}^{1:\frac{N}{T}} \leftarrow$ OpenMP_Cumulative_Sum($\tilde{\mathbf{w}}, N, T,$ Inclusive)
3: $u \sim$ Uniform[0,1)
4: #pragma omp parallel for schedule(static) num_threads($T$)
5: for $i \leftarrow 0$; $i < n$; $i \leftarrow i+1$ do
6:     $\mathbf{ncopies}^i \leftarrow \lceil \mathbf{cdf}^i + \tilde{\mathbf{w}}^i - u \rceil - \lceil \mathbf{cdf}^i - u \rceil$
7: end for
```

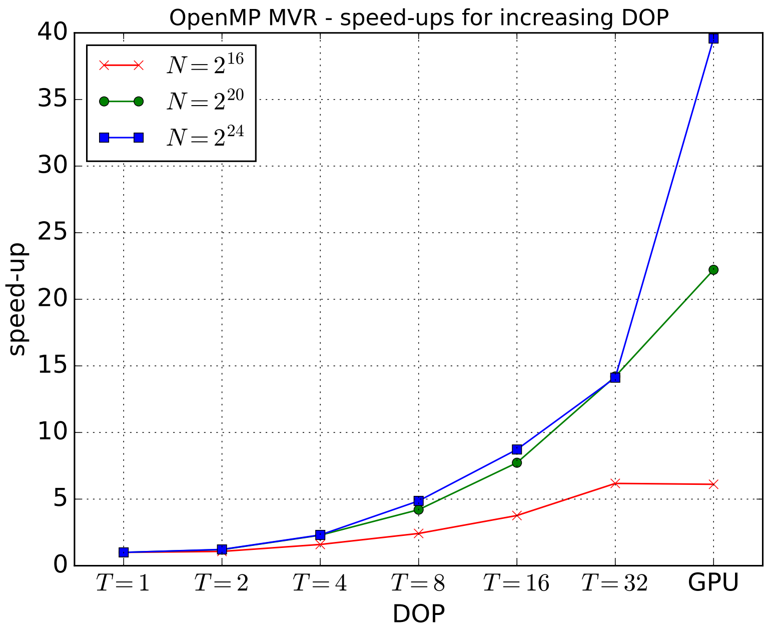


FIGURE 5.4: OpenMP Normalise - speed-ups for increasing N and DOP

the particles to duplicate. Each i-th particle to copy can indeed be found by using Binary Search over **csum** to search for the first index j such that $\mathbf{csum}^j \geq i$. While it is not explicit in [52, 53, 62], to infer **csum** it is unnecessary to perform again parallel Cumulative Sum, as one could recycle the information in **cdf** and apply (2.18) as follows:

$$\mathbf{csum}^i = \sum_{j=0}^{i} \mathbf{ncopies}^j = \lceil \mathbf{cdf}^{i+1} - u \rceil - \lceil \mathbf{cdf}^0 - u \rceil \tag{5.1}$$

Algorithm 38 briefly summarises these steps. The time complexity is $O(\frac{N}{T} \log_2 N)$ as each core needs to perform up to N Binary Searches. When N is large, due to accuracy constraints, the workload on each core is significant. As we will see in Section 5.2.5, Algorithm 38 hardly shows any speed-up vs S-R. This section develops a novel parallel redistribution that only requires one Binary Search per thread and takes advantage of the fast constant time of S-R.

Algorithm 38 Reference Parallel Redistribute

Input: N, **ncopies**, **x**, **cdf**, u, T

Output: $\mathbf{x}_{new}$

```
1: #pragma omp parallel for num_threads($T$)
2: for $i \leftarrow 0$; $i < N$; $i \leftarrow i + 1$ do
3:     $j \leftarrow$Binary Search($\mathbf{cdf}, \mathbf{ncopies}, u, i$), search
           for the first $j$ such that $\mathbf{csum}^j \geq i$, see (5.1)
4:     $\mathbf{x}^i_{new} \leftarrow \mathbf{x}^j$
5: end for
```

S-R cannot be parallelised directly as the workload could be unevenly distributed. Therefore, the proposed approach is to first balance the workload across the T cores, such that S-R could be then invoked with a fast and scalable time complexity equal to $O(\frac{N}{T})$. To do that, one needs to fully exploit the information which is stored in **csum** and **ncopies**. Although S-R has already been introduced in Algorithm 2, for this OpenMP parallel redistribution it is more convenient to use an alternative $O(N)$ implementation of S-R, illustrated in Algorithm 39. This variant differs from Algorithm 2 for two reasons: the external for loop is replaced by a while loop; the copies can be placed to the output array starting from a given index *base*.

Algorithm 39 Alternative Sequential Redistribute (S-R)

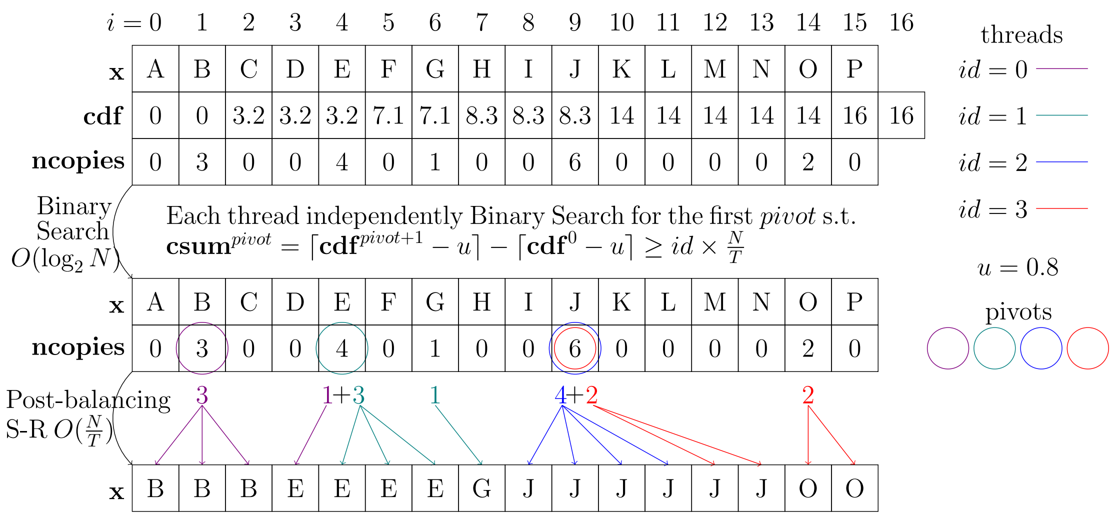
Input: N, **ncopies**, **x**, *base*

Output: $\mathbf{x}_{new}$

```
1: $i \leftarrow base$, $j \leftarrow 0$, $base$ must be 0 on a single core run
2: while $i < N$ do
3:     for $k \leftarrow 0$; $k < \mathbf{ncopies}^j$; $k \leftarrow k + 1$ do
4:         $\mathbf{x}^i_{new} \leftarrow \mathbf{x}^j$
5:         $i \leftarrow i + 1$
6:     end for
7:     $j \leftarrow j + 1$
8: end while
```

Each thread (uniquely identified by an integer $0 \leq id \leq T - 1$) has to generate $\frac{N}{T}$ particle copies, given the instructions provided by **ncopies**. However, since **ncopies** complies by definition with (2.17), it is always possible to identify T indexes, called pivots, between which the workload across the cores is equally divided. To also have a balanced data partitioning, each thread will have to place its N/T copies starting from index $i = id \times \frac{N}{T}$ in the output array. Therefore, each thread's *pivot* is the first index such that $\mathbf{csum}^{pivot} \geq id \times \frac{N}{T}$.

The threads can simultaneously find their pivot by calling Binary Search once. Since multiple threads may happen to share the same pivot, each thread must figure out how many copies of the particle $\mathbf{x}^{pivot}$ it has to create. This can be computed in constant time as the min value between $\mathbf{csum}^{pivot} - id \times \frac{N}{T}$ and $\frac{N}{T}$. That is because if two or more threads share the same pivot, only the thread with the highest *id* must create less than $\frac{N}{T}$ copies of $\mathbf{x}^{pivot}$. After that, the workload is balanced and the threads can



$i = 0$ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
threads
$id = 0$
$id = 1$
$id = 2$
$id = 3$
x A B C D E F G H I J K L M N O P
cdf 0 0 3.2 3.2 3.2 7.1 7.1 8.3 8.3 8.3 14 14 14 14 14 16 16
ncopies 0 3 0 0 4 0 1 0 0 6 0 0 0 0 2 0
Binary Search $O(\log_2 N)$
Each thread independently Binary Search for the first *pivot* s.t.
$\mathbf{csum}^{pivot} = \lceil \mathbf{cdf}^{pivot+1} - u \rceil - \lceil \mathbf{cdf}^0 - u \rceil \geq id \times \frac{N}{T}$
$u = 0.8$
pivots
x A B C D E F G H I J K L M N O P
ncopies 0 3 0 0 4 0 1 0 0 6 0 0 0 0 2 0
Post-balancing S-R $O(\frac{N}{T})$
3
1+3
1
4+2
2
x B B B E E E E G J J J J J J O O

FIGURE 5.5: Novel redistribute - Example for $N = 16$ and $T = 4$.

independently produce their $\frac{N}{T}$ particle copies by calling S-R. Since Binary Search and S-R are performed once, we can infer that the time complexity is $O(\frac{N}{T} + \log_2 N)$.

Figure 5.5 illustrates a practical example for $N = 16$ and $T = 4$ and Algorithm 40 provides an OpenMP-like description of this novel parallel redistribute.

Algorithm 40 Novel OpenMP Redistribute

Input: N, **ncopies**, **x**, **cdf**, u, T

Output: $\mathbf{x}_{new}$

```
 1: if $T == 1$ then
 2:     $\mathbf{x}_{new} \leftarrow$ Alternative S-R$(N, \mathbf{ncopies}, \mathbf{x}, 0)$, see Algorithm 39
 3: else
 4:     #pragma omp parallel num_threads$(T)${
 5:         $id \leftarrow$omp_get_thread_num()
 6:         $base \leftarrow id \times \frac{N}{T}$
 7:         $pivot \leftarrow$ Binary Search$(\mathbf{cdf}, \mathbf{ncopies}, u, base)$, search for the first
              $pivot$ such that $\mathbf{csum}^{pivot} \geq base$, see (5.1)
 8:         $n \leftarrow \min(\mathbf{csum}^{pivot} - base, \frac{N}{T})$, see (5.1) for $\mathbf{csum}^{pivot}$
 9:         $\mathbf{x}_{new}^{base}, \mathbf{x}_{new}^{base+1}, \ldots, \mathbf{x}_{new}^{base+n-1} \leftarrow \mathbf{x}_{new}^{pivot}$
10:         $\mathbf{x}_{new} \leftarrow$ Alternative S-R$(\frac{N}{T} - n, \mathbf{ncopies} + pivot, \mathbf{x} + pivot, pivot + 1)$
11:     }
12: end if
```

5.2.5 Numerical Results

This section first compares single iterations of Algorithms 38 and 40 and then two PFs working on the same model, both running $T_{PF} = 10$ iterations but differing for the constituent parallel redistribute. The two PFs are compared in the worst-case scenario which occurs when IS is relatively fast and resampling is invoked at every iteration for both PFs. Here, it is denoted that in other applications, where IS (which scales more quickly than redistribute) is slower than here, redistribution will always become the bottleneck for some bigger T.

To generate the worst-case scenario, the same benchmark stochastic volatility model as in Chapters 3 and 4 is used. The particles are initially drawn from the prior distribution. Equation (2.13) becomes $\mathbf{w}_t^i = \mathbf{w}_{t-1}^i p\left(\mathbf{Y}_t|\mathbf{x}_t^i\right)$ since the dynamics is again used as the proposal.

The two parallel OpenMP redistribution steps have been tested on the same inputs generated from a PF working on the model in (3.6a) and (3.6b). Figure 5.6a shows their scalability for $N = 2^{24}$ and increasing degree of parallelism, while Figure 5.6b shows the most significant run-times on CPU and GPU for increasing N. In Figure 5.6a, we can observe that Algorithm 38 for $T < 32$ provides little to no speed-up vs Algorithm 40 on a single core (i.e. S-R). However, Algorithm 40 achieves substantial speed-up for any $T > 1$. In Figure 5.6b, we can see that using a GPU in place of a CPU gets more effective as N increases, since the host-to-device transfer time is dominant over the computation for small N. Overall, both Figures 5.6a and 5.6b highlight that Algorithm 40 is up to six times faster than Algorithm 38 on CPU and GPU. Algorithm 40 on GPU also gives about a three-fold speed-up vs its CPU best run-time.

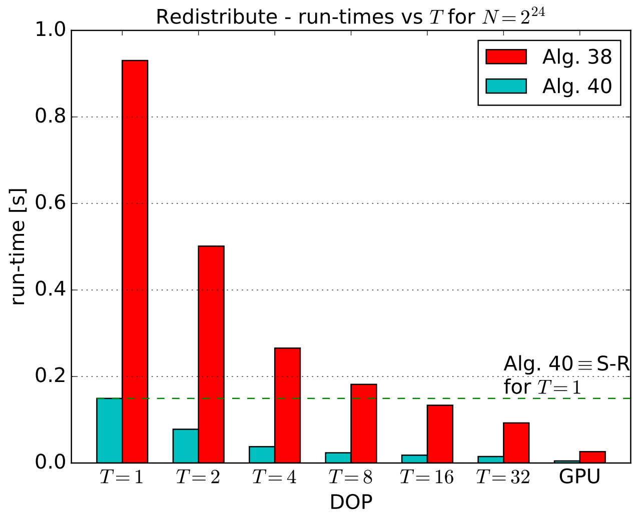
Figure 5.6c shows the results for the PFs using either Algorithm 38 or 40 and Figure 5.7 illustrates how much run-time is taken up by each task for $T_{PF} = 10$ iterations, for $T = 32$ cores and on GPU. We can see that, while Algorithm 38 still accounts for 63% of the whole run-time, Algorithm 40 is no longer the bottleneck. The overall performance of the PF has improved by up to a factor of 2.1. Here it is specified that the theoretical maximum speed-up is $1/(1 - 0.63) = 2.7$ for a 63% bottleneck, according to Amdahl's law [41]. On GPU, we can observe again over a three-fold speed-up vs the best run-time on CPU.

5.3 A Hybrid MPI+OpenMP $O(\log_2 N)$ Particle Filter

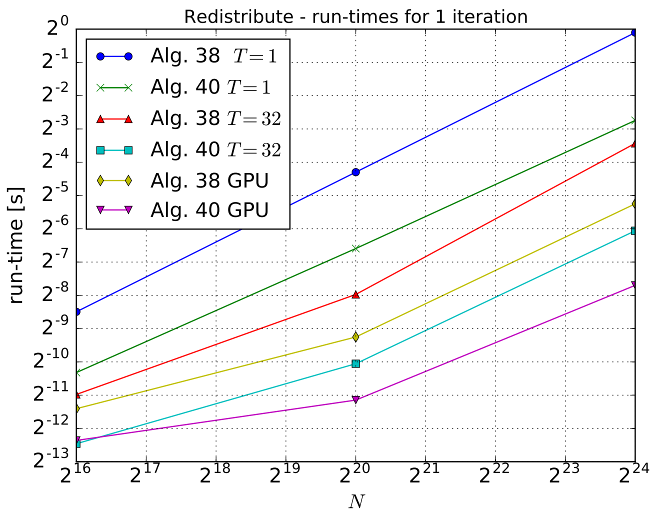
This section describes how to mix the MPI algorithms for all PF tasks, including RoSS, with the OpenMP algorithms described in the previous section. One section is again dedicated to each parallelisation strategy that is used for all PF task.

5.3.1 Embarrassingly Parallel

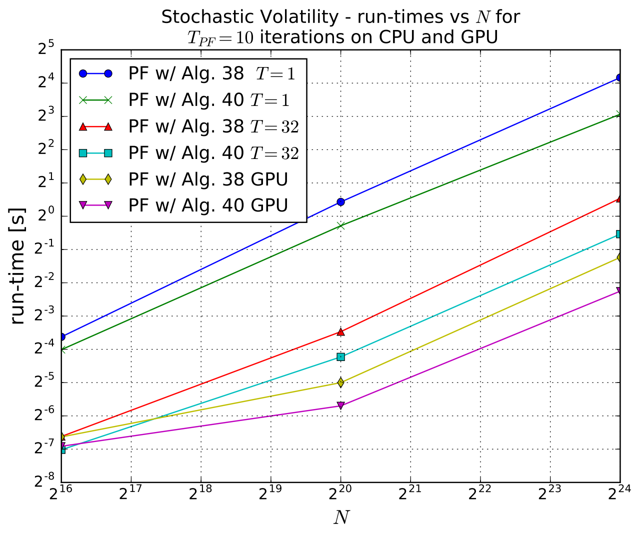
Let N be the number of particles across all MPI ranks, P be the number of MPI ranks and T the number of shared memory threads per rank. An MPI+OpenMP parallelisation for all embarrassingly parallel algorithms can be achieved by directly calling the OpenMP versions of the same in the code of each MPI rank. As long as the number of particles per MPI rank is $n = \frac{N}{P}$, the achieved time and space complexities are once again $O(1)$ for $P \times T = N$ parallel threads. Therefore, the pseudo-code for every MPI+OpenMP embarrassingly parallel task, such as `Reset` and IS, are the same as in Section 5.2.1.



(A) Redistribute - best run-times for increasing DOP



(B) Redistribute - best run-times vs N



(C) PF - best run-times vs N

FIGURE 5.6: Stochastic Volatility - run-times on OpenMP 4.5 for increasing N and DOP

5.3.2 Reduction

An MPI+OpenMP version of `Normalise`, `ESS` and `Mean` can be straightforwardly derived by calling `MPI_Allreduce` after the first for loop in Algorithms 33, 34 and 35. Then, the result of `MPI_Allreduce` is used in place of *local_sum* for `Normalise` and `ESS` or **local_sum** for `Mean`. The inputs given to `MPI_Allreduce` are the same as in MPI version of the same algorithms in Section 3.3.2. Therefore, the MPI+OpenMP pseudo-codes of these tasks are omitted for brevity.

5.3.3 Cumulative Sum

The OpenMP parallel Cumulative Sum in Algorithm 36 can be extended to a hybrid MPI+OpenMP implementation by including `MPI_Exscan` after the second for loop. This is done to keep count of the Cumulative Sum in the lower ranked MPI nodes when initialising the final for loop.

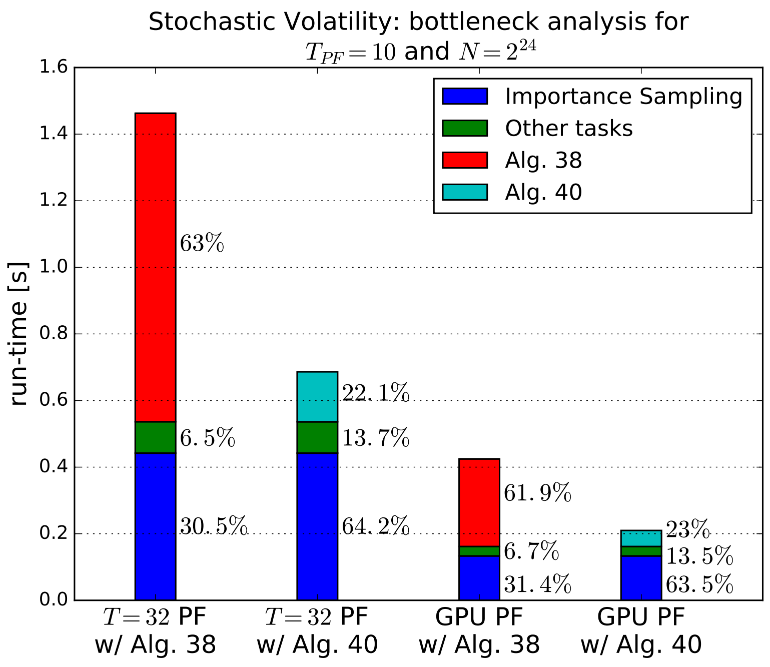


FIGURE 5.7: Stochastic Volatility - bottleneck analysis on OpenMP 4.5 for $N = 2^{24}$.

Algorithms 41 and 42 illustrate an MPI+OpenMP pseudo-code for Cumulative Sum and MVR.

5.3.4 Rotational Nearly Sort and Split

In the RoSS redistribute described in Chapter 4, all tasks except S-NS, S-R and Cumulative Sum are embarrassingly parallel. For example, Equations (4.12), (4.16) and the internal rotations during the leaf stages are embarrassingly parallel. Therefore, for

Algorithm 41 Hybrid Cumulative Sum

Input: **array**, N, T, *type*, *form*, *comm*

Output: **csum**

```
 1: #pragma omp parallel num_threads($T$){
 2:     $id \leftarrow$omp_get_thread_num(), $base \leftarrow id \times \frac{N}{T}$
 3:     $\mathbf{csum}^{base} \leftarrow \mathbf{array}^{base}$
 4:     for $i \leftarrow base + 1$; $i < base + n$; $i \leftarrow i + 1$ do
 5:         $\mathbf{csum}^{i} \leftarrow \mathbf{csum}^{i-1} + \mathbf{array}^{i}$
 6:     end for
 7:     $\mathbf{sum}^{id} \leftarrow \mathbf{csum}^{base+\frac{N}{T}-1}$
 8:     $\mathbf{coreSum}^{id} \leftarrow \mathbf{csum}^{base+\frac{N}{T}-1}$
 9: }
10: for $dist \leftarrow 1$; $dist < T$; $dist \leftarrow dist \times 2$ do
11:     #pragma omp parallel num_threads($T$){
12:         $id \leftarrow$omp_get_thread_num(), $partner \leftarrow id \oplus dist$
13:         $\mathbf{sum}^{id} \leftarrow \mathbf{sum}^{id} + \mathbf{sum}^{partner}$
14:         if $id > partner$ then
15:             $\mathbf{coreSum}^{id} \leftarrow \mathbf{coreSum}^{id} + \mathbf{sum}^{partner}$
16:         end if
17:     }
18: end for
19: MPI_Exscan($\mathbf{coreSum}^0, \mathbf{nodeSum}, 1, type,$ MPI_SUM$, comm$)
20: #pragma omp parallel num_threads($T$){
21:     $id \leftarrow$omp_get_thread_num(), $base \leftarrow id \times \frac{N}{T}$
22:     if $form ==$ Inclusive then
23:         $\mathbf{csum}^{base} \leftarrow \mathbf{nodeSum} + \mathbf{coreSum}^{id} - \mathbf{csum}^{base+\frac{N}{T}-1} + \mathbf{array}^{base}$
24:     else
25:         $\mathbf{csum}^{base} \leftarrow \mathbf{nodeSum} + \mathbf{coreSum}^{id} - \mathbf{csum}^{base+\frac{N}{T}-1}$
26:     end if
27:     for $i \leftarrow base + 1$; $i < base + n$; $i \leftarrow i + 1$ do
28:         $\mathbf{csum}^{i} \leftarrow \mathbf{csum}^{i-1} + \mathbf{array}^{i}$
29:     end for
30: }
```

these single-node MPI tasks an MPI+OpenMP parallelisation can be straightforwardly derived by adding `#pragma omp parallel for` directives above the related for loops, as explained in Sections 5.2.1.

Algorithm 41 in the previous section illustrates how to parallelise the Cumulative Sum that is required in Rotational Split.

S-R can be parallelised on OpenMP by using Algorithm 40. This SMA parallelisation requires knowledge of the Cumulative Sum over **ncopies**. As mentioned in Section 5.2.4, on SMA only this can be inferred with little to no overhead from the CDF of the weights as in Equation (5.1). However, this is not true on MPI+OpenMP, since the elements of **ncopies** after MVR will be completely changed by the load-balancing phases in RoSS. The Cumulative Sum **csum** which is computed and updated during Rotational Split cannot be used either, because correctness is not guaranteed for the i-th indexes such

Algorithm 42 Minimum Variance Resampling (MVR)

```
Input: $\tilde{\mathbf{w}}$, $N$, $P$, $T$, $p$
Output: ncopies
 1: $n \leftarrow \frac{N}{P}$
 2: $comm \leftarrow$ MPI_COMM_WORLD
 3: if $p == 0$ then
 4:     $\mathbf{cdf}^0 \leftarrow 0$, because $\mathbf{cdf} \in \mathbb{R}^{N+1}$
 5: end if
 6: if $p == 0$ then
 7:     $\mathbf{cdf}^{1:n} \leftarrow$ Hybrid_Cumulative_Sum($\tilde{\mathbf{w}}, n, T,$ MPI_DOUBLE, Inclusive, $comm$)
 8: else
 9:     $\mathbf{cdf}^{1:n-1} \leftarrow$ Hybrid_Cumulative_Sum($\tilde{\mathbf{w}}, n, T,$ MPI_DOUBLE, Inclusive, $comm$)
10: end if
11: $u \sim$ Uniform[0,1)
12: MPI_Bcast($u, 1,$ MPI_DOUBLE$, 0, comm$), broadcast $u$ to other cores in $O(\log_2 P)$
13: #pragma omp parallel for schedule(static) num_threads($T$)
14: for $i \leftarrow 0$; $i < n$; $i \leftarrow i + 1$ do
15:     $\mathbf{ncopies}^i \leftarrow \lceil \mathbf{cdf}^i + \tilde{\mathbf{w}}^i - u \rceil - \lceil \mathbf{cdf}^i - u \rceil$
16: end for
```

that $\mathbf{ncopies}^i = 0$. This means that Binary Search cannot be used yet as **csum** is not sorted. An extra computation of Cumulative Sum to correct **csum** is therefore necessary before calling Algorithm 40.

S-NS can be performed in $O(\frac{N}{T} + \log_2 T)$ computations by using a similar strategy to Rotational Nearly Sort. Here, each thread *id* is tasked with working on the memory space of the output array between index $id \times \frac{N}{T}$ and index $(id + 1) \times \frac{N}{T} - 1$. The threads start nearly sorting in $O(\frac{N}{T})$ by calling Algorithm 22 over their iteration space. This is done to also compute $\mathbf{zeros}^{id} \in \mathbb{Z}$, the number of zeros between those indexes. After that, $\mathbf{shifts}^{id} \in \mathbb{Z}$, the number of positions that each particle must rotate by, is computed in $O(\log_2 T)$ as in Equation (4.6). At this point, instead of performing the rotations by using a LSB-to-MSB strategy as in Algorithm 25, on OpenMP it is possible to rotate in $O(\frac{N}{T})$. Every particles i such that $\mathbf{ncopies}^i > 0$ can indeed be copied to index $i - \mathbf{shifts}^{id}$ of the output array without risking collisions. To ensure data coherency without using synchronisation points, a temporary array must be used during the copying phase.

5.3.5 Numerical Results

This section first compares single iterations of RoSS redistribute on MPI with its equivalent on a hybrid memory architecture using MPI and OpenMP. Then it compares an MPI version of a PF working on the stochastic volatility model with its equivalent implementation on MPI+OpenMP. $N = \{2^{16}, 2^{20}, 2^{24}\}$ as in all experiments in this thesis. Up to 256 parallel cores are employed. On hybrid memory, the results for 2 cores are always equivalent to the same on MPI, as these two cores are mapped to two distributed computing nodes. For brevity reasons, the reported results on hybrid memory for more than 2 cores only refer to the best combination of P and T, respectively the number of MPI ranks and the number of parallel threads within the same MPI rank. Each reported run-time is again the median of 20 runs using the same N, P, T triples.

Figure 5.8 compares the results for RoSS on MPI with its equivalent implementation on MPI+OpenMP. As we can see, the hybrid memory version, can only provide optimisation for $N = 2^{16}$ particles and only when the MPI version has stopped scaling already. Also, the best results for the hybrid memory implementation are for the most part for a low $T \leq 4$. After a thorough investigation on the profiler, it appears that the highly volume of memory-copies and cache misses during the internal rotations of the two leaf stages do not offer good scalability, especially in comparisons with their equivalent on MPI. To some extent, this also nullifies the good speed-up that other embarrassingly parallel tasks offer, such as (4.12) and (4.16).

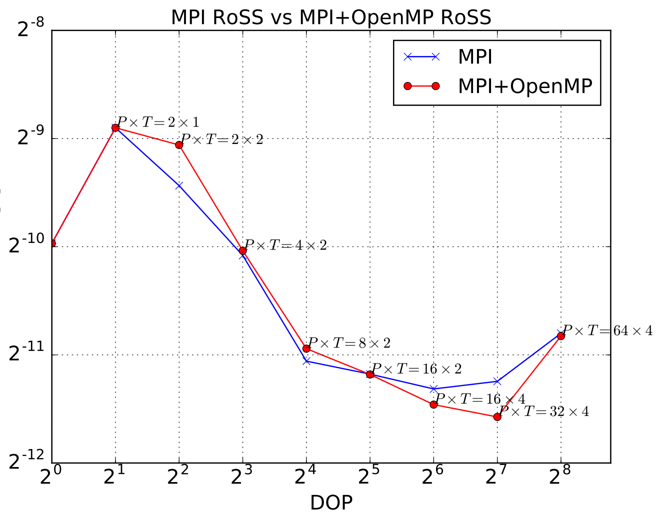
Broadly speaking, these results are expected: apart from the specific side effects of mixing MPI and OpenMP in this particular example, this behaviour is actually typical when it comes to comparisons between MPI and MPI+OpenMP on CPUs [15, 16], especially for strongly memory bounded tasks such as redistribute. The general assumption is that, when P increases, the memory gets more distributed, reducing the frequency of the caching problems. In this case, increasing P further may cause saturation due to the communication, and increasing T may be a more sensible option.

The results for the stochastic volatility model are provided in Figure 5.9. Here we can see that using MPI-everywhere or MPI+OpenMP, whenever either architecture shows better performance on redistribute, does not translate into a significant speed-up for the PF. This is in part explained from the results in Figure 5.9, but mostly from the outcomes in Section 4.4.5, where it is explained that RoSS has not yet emerged as bottleneck over IS, see Figure 4.10. Therefore, we can conclude that, for the same DOP of this experiment, the efforts should be re-orientated towards investigating solutions to optimise IS. Some ideas to achieve that are explained in the next section.

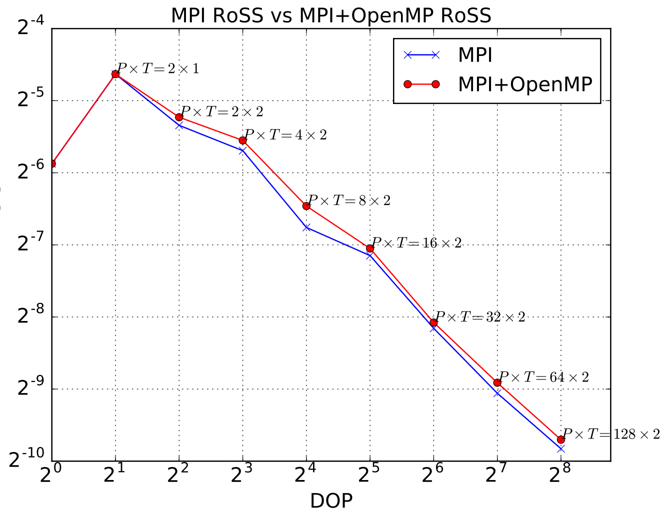
5.4 Conclusions

This chapter is divided into two parts. In the first one, a novel parallel redistribute for SMAs has been introduced. The achieved time complexity is $O(\frac{N}{T} + \log_2 T)$ for T shared memory threads. The chosen programming model is OpenMP 4.5, one of the most popular shared-memory APIs which supports both mainstream CPUs and GPU offload, starting from version 4.5 on. Therefore, the results for this first part are provided both on CPU and GPU, using respectively a 32-core machine and a Tesla V100 graphics card. Here, it is shown that the proposed approach is up to six times faster than a referenced one which scales as $O(\frac{N}{T} \times \log_2 T)$.

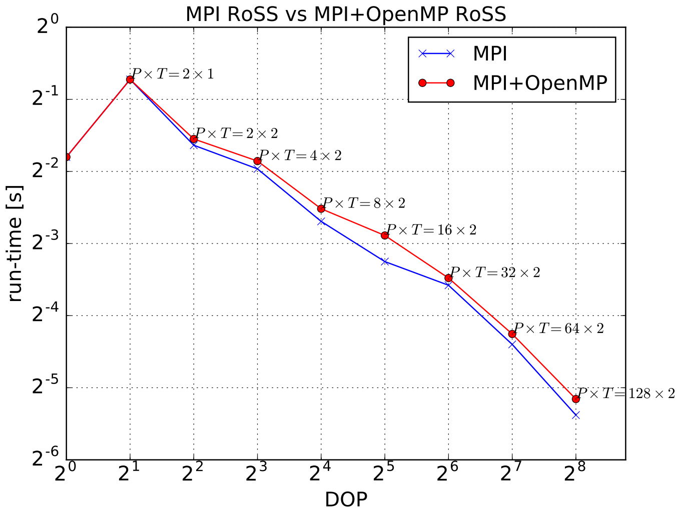
In the second part, the algorithms presented in Chapter 4 on MPI have been translated for hybrid memory architectures by mixing MPI and OpenMP. In this case, only the CPUs are employed as the cluster that has been used for this thesis mounts one GPU



(A) $N = 2^{16}$

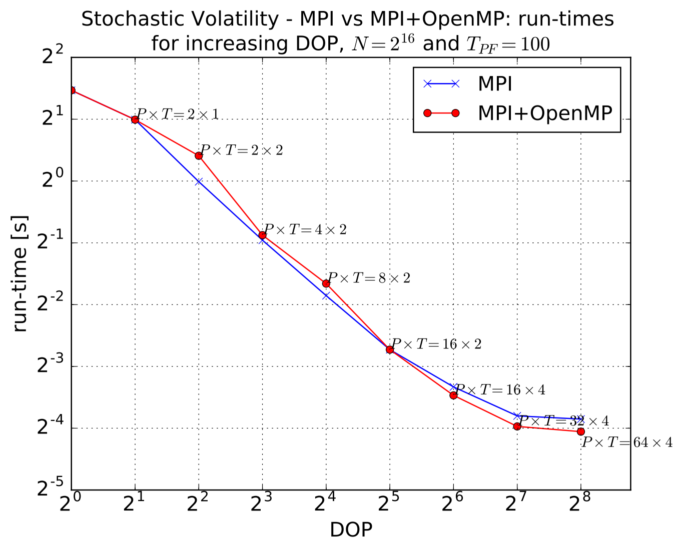


(B) $N = 2^{20}$

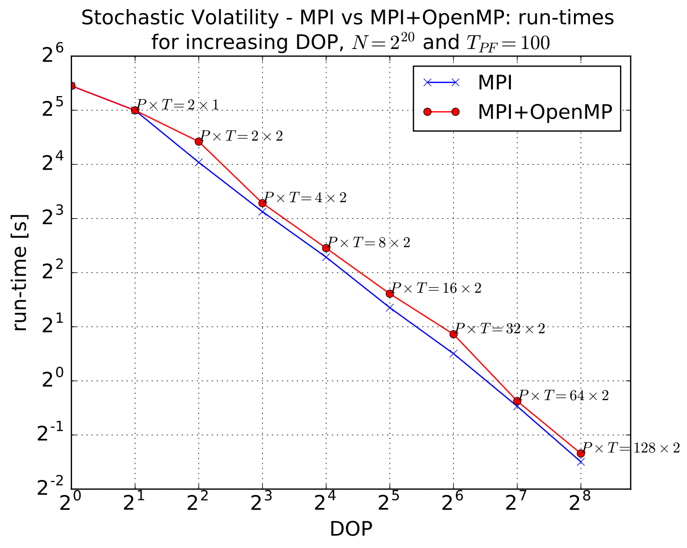


(C) $N = 2^{24}$

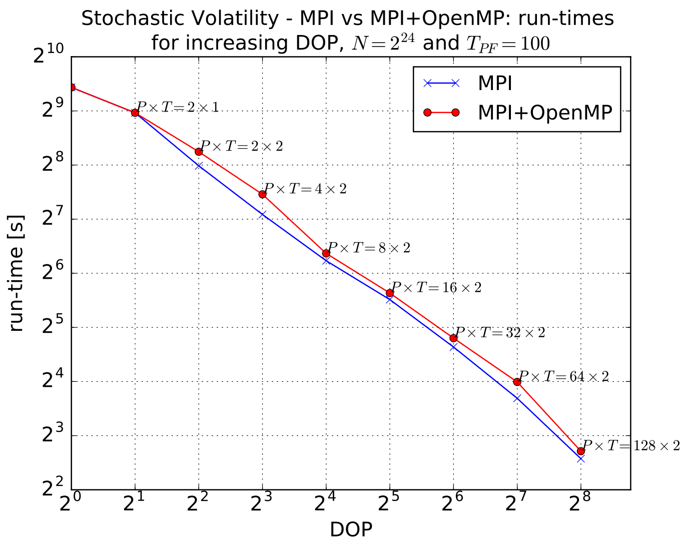
FIGURE 5.8: MPI+OpenMP RoSS vs MPI RoSS - run-times for increasing N and P



(A) $N = 2^{16}$



(B) $N = 2^{20}$



(C) $N = 2^{24}$

FIGURE 5.9: Stochastic Volatility MPI+OpenMP PF vs MPI PF - run-times for increasing N and P

node only. The results show that the hybrid implementation is slightly better for low values of N, when the MPI-everywhere alternative has stopped scaling, while using MPI only shows some improvements for large values of N. The user should then be able to choose the preferred version between MPI-everywhere, OpenMP-only or MPI+OpenMP, depending on the performance and the available resources. This is possible by setting $T = 1$ for MPI-everywhere, $P = 1$ for OpenMP only, or $P > 1$ and $T > 1$ for hybrid memory architectures.

With the results of this chapter and Chaper 4 in view, it is clear that efforts should now be focused more on IS than redistribute both for run-time and accuracy. One straightforward way to improve the run-time performance is to port the MPI+OpenMP implementation from CPU to GPU. In Section 5.2, it is shown that IS on GPU can be up to three times faster than its equivalent on CPU. To achieve this goal, the Big Hypothesis project is currently engaging with IBM to port the MPI+OpenMP algorithm to a 128-GPU cluster. Also, as mentioned in the previous chapter, models can be complex, e.g. requiring highly computationally intensive numerical integrators to achieve better accuracy. The next chapter shows how to use NUTS moves as proposal to improve the accuracy of the estimations. This is done by installing SMC methods into Stan, a highly popular statistical programming model which uses NUTS and a fast gradient based on auto-differentiation to sample from static models.

Chapter 6

Streaming-Stan and SMC-Stan: Two High Performance Computing Extensions for Stan

6.1 Introduction

The previous chapter describes a parallel SMC method which achieves $O(\log_2 N)$ time complexity on a hybrid memory architecture having N parallel CPU cores. This parallel algorithm has been implemented on C++ using MPI and OpenMP for parallelism, a very common choice in HPC applications. Although C++ is still used quite extensively, many other languages such as MATLAB, R and Python have been used in many research papers on SMC methods. The goal of this chapter is to make the progress previously discussed in this thesis more widely available, without having to translate the code to every other existing language. To achieve that, this chapter presents Streaming-Stan and SMC-Stan, two HPC extension packages for Stan which respectively embody the described parallel PF and SMC sampler on MPI+OpenMP, along with the already existing functionalities of Stan. Extending Stan is indeed perfectly compatible with the goal of this chapter, since Stan is a popular and intuitive statistical programming language whose back end is also written in C++, and which already interfaces with other languages, such as MATLAB, R, Python and Julia.

In doing so, the rest of this chapter is organised as follows: Section 6.2 describes SMC-Stan and the different proposal distributions which the installed SMC sampler equips, and shows the performance on an exemplary static model. Section 6.3 illustrates Streaming-Stan; in this version also, three proposal distributions are described, two of which being novel and combining a FL-SMC method with HMC and NUTS. The performance of Streaming-Stan are provided in Section 6.4 on several dynamic models under different testing conditions. Section 6.5 draws the conclusions and gives suggestions for future work. The reader is referred to Appendix B for a brief tutorial about Stan and to Appendix C for implementation details of Streaming-Stan and SMC-Stan.

6.2 SMC-Stan

Although this thesis is mostly focused on PFs, in this case it is more intuitive to describe SMC-Stan first, before presenting Streaming-Stan. This is because the novelty described in Section 6.3 also includes most of the mathematics from this section. The reader is referred to Section C.1 for all implementation details of SMC-Stan.

6.2.1 Proposal Distributions

As anticipated in the previous section and elsewhere in Chapter 2, SMC-Stan offers the possibility to run an SMC sampler on a static model, described by a .stan model, and also supports several types of proposal distributions. This section provides the mathematical details of the three proposals which are currently installed in SMC-Stan.

6.2.1.1 Random Walk

The original SMC sampler described in [61, 65] uses a random walk (RW) proposal (akin to Metropolis Hasting MCMC methods) where the N samples at each SMC iteration are simply moved in space by adding a random vector $\mathcal{N}(\mathbf{0}, \mathbf{1})$ to each sample. The proposed samples are always accepted, weighted as in (2.23) and resampling is tasked with accepting or rejecting the new samples.

6.2.1.2 HMC

As suggested in Section 2.2, better MCMC moves than RW, such as HMC and NUTS, can be used instead. It is possible to prove that using HMC to propose a new sample $\mathbf{x}_t^i$ given $\mathbf{x}_{t-1}^i$ requires the following proposal distribution:

$$q(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i) = \frac{q_{\mathbf{v}}(\mathbf{v}_t^i)}{\left|\left|\frac{\partial \mathbf{f}(\mathbf{x}_{t-1}^i, \mathbf{v}_{t-1}^i)}{\partial \mathbf{v}}\right|\right|} = \frac{q_{\mathbf{v}}(\mathbf{v}_t^i)}{||\mathbf{J}(\mathbf{x}_{t-1}^i, \mathbf{v}_{t-1}^i)||} \tag{6.1}$$

where $q_{\mathbf{v}}()$ is the proposal distribution of the momentum vector $\mathbf{v}$, $\mathbf{f}$ is a numerical integrator such that $\mathbf{f}(\mathbf{x}_{t-1}^i, \mathbf{v}_{t-1}^i) = \mathbf{x}_t^i, \mathbf{v}_t^i$, and $\mathbf{J}$ is the Jacobian matrix related to the same numerical integrator. Equation (2.23) also needs a backward kernel, which is commonly chosen as the reverse proposal $q(\mathbf{x}_{t-1}^i|\mathbf{x}_t^i)$, such that in this case it is equal to:

$$L(\mathbf{x}_{t-1}^i|\mathbf{x}_t^i) = \frac{q_{\mathbf{v}}(\tilde{\mathbf{v}}_t^i)}{\left|\left|\frac{\partial \mathbf{f}(\mathbf{x}_t^i, \tilde{\mathbf{v}}_t^i)}{\partial \mathbf{v}}\right|\right|} = \frac{q_{\mathbf{v}}(\tilde{\mathbf{v}}_t^i)}{||\mathbf{J}(\mathbf{x}_t^i, \tilde{\mathbf{v}}_t^i)||} \tag{6.2}$$

where $\tilde{\mathbf{v}}_t^i$ is a momentum vector such that $\mathbf{f}(\mathbf{x}_t^i, \tilde{\mathbf{v}}_t^i) = \mathbf{x}_{t-1}^i, \mathbf{v}_{t-1}^i$. Computing $\tilde{\mathbf{v}}_t^i$ is not always easy, since it involves inverting the numerical integrator $\mathbf{f}$. However, Leapfrog, the most common numerical integrator for HMC, is time reversible, a property such that:

$$\mathbf{f}(\mathbf{x}_{t-1}^i, \mathbf{v}_{t-1}^i) = \mathbf{x}_t^i, \mathbf{v}_t^i \tag{6.3a}$$

$$\mathbf{f}(\mathbf{x}_t^i, -\mathbf{v}_t^i) = \mathbf{x}_{t-1}^i, \mathbf{v}_{t-1}^i \tag{6.3b}$$

In other words, time reversibility means that if we reverse the integration from the final offset with the opposite of the final momentum, we get back where we started. Therefore, if we apply (6.1) and (6.2) to (2.23) and set $\tilde{\mathbf{v}}_t^i = -\mathbf{v}_t^i$ we obtain:

$$\mathbf{w}_t^i = \mathbf{w}_{t-1}^i \frac{p(\mathbf{x}_t^i|\mathbf{Y})}{p(\mathbf{x}_{t-1}^i|\mathbf{Y})} \frac{q_{\mathbf{v}}(-\mathbf{v}_t^i)||\mathbf{J}(\mathbf{x}_{t-1}^i, \mathbf{v}_{t-1}^i)||}{q_{\mathbf{v}}(\mathbf{v}_t^i)||\mathbf{J}(\mathbf{x}_t^i, -\mathbf{v}_t^i)||} \tag{6.4}$$

It is possible to prove that $||\mathbf{J}(\mathbf{x}_{t-1}^i, \mathbf{v}_{t-1}^i)|| = 1$ for Leapfrog, as shown in [98]. Furthermore, it is possible to prove that $||\mathbf{J}(\mathbf{x}_{t-1}^i, \mathbf{v}_{t-1}^i)|| = ||\mathbf{J}(\mathbf{x}_t^i, -\mathbf{v}_t^i)||$ for reversible numerical integrators, as shown in Appendix D. Therefore, (6.4) can be simplified to:

$$\mathbf{w}_t^i = \mathbf{w}_{t-1}^i \frac{p(\mathbf{x}_t^i|\mathbf{Y})}{p(\mathbf{x}_{t-1}^i|\mathbf{Y})} \frac{q_{\mathbf{v}}(-\mathbf{v}_t^i)}{q_{\mathbf{v}}(\mathbf{v}_t^i)} \tag{6.5}$$

which does not require computation of the Jacobian terms and their determinants which may be computationally expensive.

6.2.1.3 NUTS

As explained in Section 2.2.3, NUTS also employs Leapfrog for an adaptive number of Leapfrog steps. Hence, it is relatively straightforward to infer that the samples drawn by an SMC sampler which uses NUTS moves as proposal are again weighted as in (6.5).

6.2.2 Numerical Results

This section repeats the experiment in Section 2.2.3.1 on SMC-Stan by using an SMC sampler employing three different proposal distributions: RW, HMC and NUTS. The .stan file for this model can be found in Section B.3.2.

In SMC-Stan, the .stan file is unchanged and, while the user can still run the command in Section B.3.2 to execute MCMC, it is also possible to employ an SMC sampler using RW, HMC or NUTS proposals by running respectively the following commands:

```
$ mpirun -np 1 neals_funnel method=sample algorithm=smcs proposal=rw T=1 \
Tsmc=100 num_samples=1024
$ mpirun -np 1 neals_funnel method=sample algorithm=smcs proposal=hmc \
num_leapfrog_steps=5 stepsize=0.47 T=1 Tsmc=100 num_samples=1024
$ mpirun -np 1 neals_funnel method=sample algorithm=smcs proposal=NUTS \
stepsize=0.47 T=1 Tsmc=100 num_samples=1024
```

The three commands above run an SMC sampler for $T_{SMC} = 100$ iterations by drawing and weighting $N = 1024$ samples each time, and using $P = 1$ MPI nodes and $T = 1$ OpenMP threads.

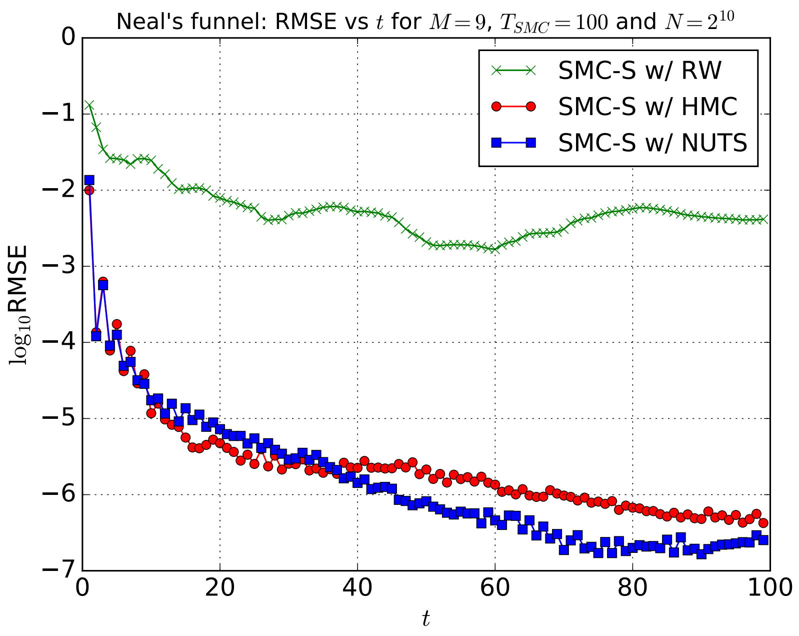


FIGURE 6.1: SMC-Stan - Neal's funnel example

Figure 6.1 shows the RMSE of the estimates in logarithmic scale, for the same SMC sampler using RW, HMC or NUTS as proposal. As we can see, using HMC or NUTS greatly improves the accuracy of the estimation with respect to a RW proposal which is struggling due to the high dimension of the state $M = 9$. However, the most important outcome of this section is that it is actually possible to install an SMC method in Stan alongside MCMC without changing Stan's syntax. To some extent, this was imaginable since MCMC and SMC samplers are designed to work on the same static model.

On the other hand, Stan does not currently provide support for dynamic models, i.e. it is not designed to use PFs. The next section presents Streaming-Stan which overcomes this limitation.

6.3 Streaming-Stan

Streaming-Stan is a novel extension package for Stan, similar to SMC-Stan, which, however, gives the user the possibility to describe dynamic models using the same syntax of regular Stan. As well as SMC-Stan, Streaming-Stan also provides several proposal distributions which are described in the following section. The reader is referred to Section C.2 for all implementation details about Streaming-Stan.

6.3.1 Proposal Distributions

This section describes FL-SMC, FL-HMC and FL-NUTS, the proposal distributions that Streaming-Stan provides for PFs and Fixed-Lag SMC methods. FL-HMC and FL-NUTS are novel and respectively use HMC and NUTS in a FL-SMC method.

6.3.1.1 Fixed-Lag SMC

As explained in Section 2.1.4, Fixed-Lag SMC methods sample and weight each particle as in Equations (2.26) a (2.28) which are repeated here for extra clarity:

$$\mathbf{x}^i_{t-l:t} \sim q(\mathbf{x}^i_{t-l:t}|\overline{\mathbf{x}}^i_{t-l-1}, \mathbf{Y}_{t-l:t})$$

$$\mathbf{w}^i_t = \mathbf{w}^i_{t-1}\frac{p(\mathbf{x}^i_{t-l:t}|\overline{\mathbf{x}}^i_{t-l-1})p(\mathbf{Y}_{t-l:t}|\mathbf{x}^i_{t-l:t})L(\overline{\mathbf{x}}^i_{t-l:t-1}|\mathbf{x}^i_{t-l:t-1})}{p(\overline{\mathbf{x}}^i_{t-l:t-1}|\overline{\mathbf{x}}^i_{t-l-1})p(\mathbf{Y}_{t-l:t-1}|\overline{\mathbf{x}}^i_{t-l:t})q(\mathbf{x}^i_{t-l:t}|\overline{\mathbf{x}}^i_{t-1}, \mathbf{Y}_{t-l:t})}$$

These equations are function of the lag l which should be given as input. Also, as explained in Sections 2.1.4 and C.2, they can respectively be coded to become equal to

$$\mathbf{x}^i_t \sim q(\mathbf{x}^i_t|\mathbf{x}^i_{t-1}, \mathbf{Y}_t)$$

and

$$\mathbf{w}^i_t = \mathbf{w}^i_{t-1}\frac{p(\mathbf{x}^i_t|\mathbf{x}^i_{t-1})p(\mathbf{Y}_t|\mathbf{x}^i_t)}{q(\mathbf{x}^i_t|\mathbf{x}^i_{t-1}, \mathbf{Y}_t)}$$

when $l = 0$, i.e. Equations (2.12) and (2.13). Therefore, choosing FL-SMC with $l = 0$ in Streaming-Stan is equivalent to running a SIR PF.

6.3.1.2 Fixed-Lag HMC

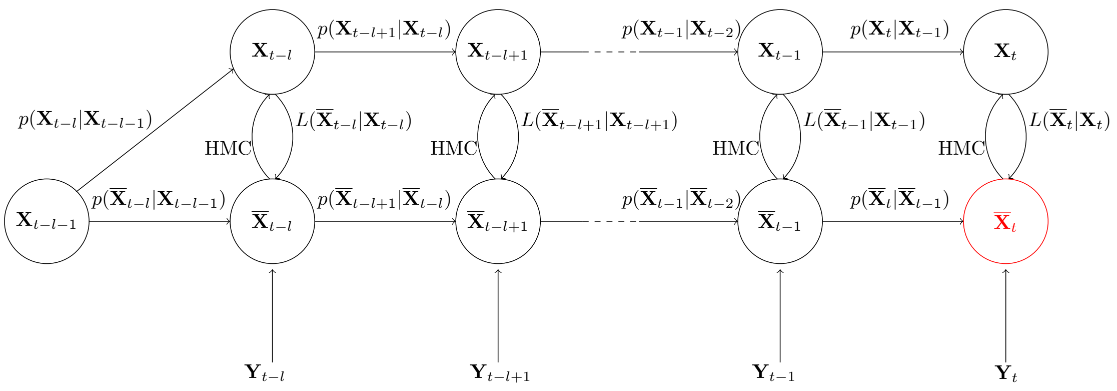

$\mathbf{X}_{t-l}$
$p(\mathbf{X}_{t-l+1}|\mathbf{X}_{t-l})$
$\mathbf{X}_{t-l+1}$
$p(\mathbf{X}_{t-1}|\mathbf{X}_{t-2})$
$\mathbf{X}_{t-1}$
$p(\mathbf{X}_t|\mathbf{X}_{t-1})$
$\mathbf{X}_t$
$p(\mathbf{X}_{t-l}|\mathbf{X}_{t-l-1})$
$L(\overline{\mathbf{X}}_{t-l}|\mathbf{X}_{t-l})$
$L(\overline{\mathbf{X}}_{t-l+1}|\mathbf{X}_{t-l+1})$
$L(\overline{\mathbf{X}}_{t-1}|\mathbf{X}_{t-1})$
$L(\overline{\mathbf{X}}_t|\mathbf{X}_t)$
HMC
HMC
HMC
HMC
$\mathbf{X}_{t-l-1}$
$p(\overline{\mathbf{X}}_{t-l}|\mathbf{X}_{t-l-1})$
$\overline{\mathbf{X}}_{t-l}$
$p(\overline{\mathbf{X}}_{t-l+1}|\overline{\mathbf{X}}_{t-l})$
$\overline{\mathbf{X}}_{t-l+1}$
$p(\overline{\mathbf{X}}_{t-1}|\overline{\mathbf{X}}_{t-2})$
$\overline{\mathbf{X}}_{t-1}$
$p(\overline{\mathbf{X}}_t|\overline{\mathbf{X}}_{t-1})$
$\overline{\mathbf{X}}_t$
$\mathbf{Y}_{t-l}$
$\mathbf{Y}_{t-l+1}$
$\mathbf{Y}_{t-1}$
$\mathbf{Y}_t$

FIGURE 6.2: Fixed-Lag Hamiltonian Monte Carlo: state flow

Both PFs and Fixed-Lag SMC suffer from the curse of dimensionality, which causes accuracy loss as the state dimension M increases. In the case of regular PF models, an alternative solution called Sequential Markov Chain Monte Carlo (SMCMC) is proposed, consisting of one or more Hamiltonian Monte Carlo (HMC) unweighted chains, sampling from a dynamic target distribution [23, 72, 78, 95]. This suggests a similar approach could be done in IS for Fixed-Lag SMC methods, which can cover both PF models and models where the measurements are time-delayed.

Considering the state flow diagram for FL-SMC in Figure 2.3, the key idea is again to replace every old particle $\overline{\mathbf{x}}_\tau^i$ with a new particle $\mathbf{x}_\tau^i$ $\forall \tau = t-l, ..., t$. However, instead of sampling $\mathbf{x}_\tau^i$ given $\mathbf{x}_{\tau-1}^i$, FL-HMC uses HMC to upgrade each $\overline{\mathbf{x}}_\tau^i$ into $\mathbf{x}_\tau^i$, such that the forward and backward kernels are reversed and the determinants of their Jacobian terms cancel, as done for Equation (6.5). However, as can be observed in in Figure 2.3, the old trajectory $\overline{\mathbf{x}}_{t-l:t-1}^i$ does not have a term at time step t from which HMC should generate $\mathbf{x}_t^i$. Therefore, FL-HMC is designed to first extend the old trajectory by randomly sampling $\overline{\mathbf{x}}_t^i$ from $\overline{\mathbf{x}}_{t-1}^i$, before calling HMC on each $\overline{\mathbf{x}}_\tau^i$ $\forall \tau = t-l, ..., t$. Figure 6.2 illustrates the state flow diagram for FL-HMC. Since the determinants of the Jacobian terms of each backward kernel vs forward kernel ratio cancel, it is relatively straightforward to infer that the weight equation becomes:

$$\mathbf{w}_t^i = \mathbf{w}_{t-1}^i \frac{p(\mathbf{x}_{t-l:t}^i|\mathbf{x}_{t-l-1}^i)p(\mathbf{Y}_{t-l:t}|\mathbf{x}_{t-l:t}^i)q_{\mathbf{v}}(-\mathbf{v}_{t-l:t}^i)}{p(\overline{\mathbf{x}}_{t-l:t}^i|\mathbf{x}_{t-l-1}^i)p(\mathbf{Y}_{t-l:t}|\overline{\mathbf{x}}_{t-l:t}^i)q_{\mathbf{v}}(\mathbf{v}_{t-l:t}^i)} \tag{6.6}$$

6.3.1.3 Fixed-Lag NUTS

The Fixed-Lag NUTS proposal is equivalent to Fixed-Lag HMC, but NUTS is used in place of HMC. Therefore, the state flow diagram in Figure 6.2 is still valid, along with the importance weight equation (6.6).

6.4 Numerical Results

This section shows the performance of Fixed-Lag HMC and Fixed-Lag NUTS, the most advanced proposal distributions of Streaming-Stan, under four different testing conditions: flexibility on varying dynamic models (see Section 6.4.1), accuracy for increasing dimension of $\mathbf{X}_t$ and $\mathbf{Y}_t$ (see Section 6.4.2), accuracy on long-term memory models (see Section 6.4.3), and run-time for increasing number of particles (see Section 6.4.4).

6.4.1 Flexibility

The goal of section is to show the flexibility of Fixed-Lag HMC and Fixed-Lag NUTS when it comes to work on a relatively large portfolio of dynamic models, which previous research has already shown that can be effectively solved using RW proposal (i.e. FL-SMC with $l = 0$). Therefore, this section aims to show that Fixed-Lag HMC and Fixed-Lag NUTS performs at least equally to RW on these models. The reasons and scenarios as to why and when Fixed-Lag HMC and Fixed-Lag NUTS may perform better than FL-SMC will be analysed in details in Sections 6.4.2 and 6.4.3.

6.4.1.1 Stochastic Volatility

The first example is the Stochastic Volatility model which is first illustrated in this thesis in Section 3.4.5.2. The Streaming-Stan code for this model can be found in Section C.2.5.1, along with the compiling instructions.

The following table shows the average RMSE in logarithmic scale.

Symbol [unit]	FL-SMC	FL-HMC	FL-NUTS
$\mathbf{X}$ [m]	−0.345	−0.342	−0.343

TABLE 6.1: Stochastic Volatility: average RMSE in $\log_{10}$ scale of the state over $T_{PF} =$ 100 time steps and for $N = 2^9$ particles.

As we can see, the performance are comparable for all the different proposals.

6.4.1.2 Flood Water Level

The second example is a popular model to monitor the flood water level in an urban area. This model is found in several publications such as [6, 73, 74, 86] and is described by the following equations:

$$\mathbf{X}_t = 0.5\mathbf{X}_{t-1} + \frac{25\mathbf{X}_{t-1}}{1+(\mathbf{X}_{t-1})^2} + \cos(1.2t) + \mathcal{N}(0,1) \tag{6.7a}$$

$$\mathbf{Y}_t = 0.05(\mathbf{X}_{t-1})^2 + \mathcal{N}(0,1) \tag{6.7b}$$

where $\mathbf{X}_0 \sim \mathcal{N}(0,1)$.

The Streaming-Stan code for this model is found in Section C.2.5.2.

The state estimation accuracy, expressed as average RMSE in $\log_{10}$ scale, is shown in the following table:

Symbol [unit]	FL-SMC	FL-HMC	FL-NUTS
$\mathbf{X}$ [m]	0.343	0.352	0.381

TABLE 6.2: Flood Water Level: average RMSE in $\log_{10}$ scale of the state over $T_{PF} =$ 100 time steps and for $N = 2^9$ particles.

As we can see, there is no significant difference between the different proposals.

6.4.1.3 Bearing-Only Tracking

The next example is a popular four-dimensional state dynamic model for bearing-only tracking, where the state is represented by the position and velocity of the tracked object which are both two-dimensional physical quantities. This model was previously presented in several publications, such as in [6], and used in [29] to test the Block Sampling SIR PF. In accordance with [29], the state is composed of four elements denoted such that $\mathbf{X}_t = [p_x, v_x, p_y, v_y]$ where $\mathbf{X}_{t,0} = p_x$, $\mathbf{X}_{t,2} = p_y$ represent the state position and $\mathbf{X}_{t,1} = v_x$, $\mathbf{X}_{t,3} = v_y$ represent the state velocity. The model is defined as follows:

$$\mathbf{X}_t = \mathbf{A} \cdot \mathbf{X}_{t-1} + \mathbf{V}_t \tag{6.8a}$$

$$\mathbf{Y}_{t,k} = \arctan\left(\frac{\mathbf{X}_{t,2} - \mathbf{p}_{k,2}}{\mathbf{X}_{t,0} - \mathbf{p}_{k,0}}\right) + \mathbf{W}_{t,k}, \quad \forall k = 1, \ldots, M_y \tag{6.8b}$$

where $(\mathbf{p}_{k,0}, \mathbf{p}_{k,1})$ are the Cartesian coordinates of the k-th sensor.

$$\mathbf{A} = \begin{bmatrix} 1 & \Delta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \Delta \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{\Sigma} = q \begin{bmatrix} \frac{\Delta^3}{3} & \frac{\Delta^2}{2} & 0 & 0 \\ \frac{\Delta^2}{2} & \Delta & 0 & 0 \\ 0 & 0 & \frac{\Delta^3}{3} & \frac{\Delta^2}{2} \\ 0 & 0 & \frac{\Delta^2}{2} & \Delta \end{bmatrix}$$

The process noise is $\mathbf{V}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{\Sigma})$ and each sensor noise is $\mathbf{W}_{t,k} \sim \mathcal{N}(0, 0.333)$. The initial state $\mathbf{X}_0$ has the identity matrix as covariance and mean equal to the true initial simulated point of the system. The parameter Δ represents the sampling period which is set to $\Delta = 1$ s, and the scalar $q \in \mathbb{R}$ is set to 5.0.

The Streaming-Stan code for this model is found in Section C.2.5.3.

The average RMSEs (again in $\log_{10}$ scale) of position and velocity over $T_{PF} = 100$ time steps are shown in Table 6.3. which shows that Fixed-Lag HMC and Fixed-Lag

Symbol [unit]	**FL-SMC**	**FL-HMC**	**FL-NUTS**
(p_x, p_y) [m]	-1.0154	-1.1892	-1.1915
(v_x, v_y) [m/s]	-0.6978	-0.8135	-0.8446

Table 6.3: Bearing-Only Tracking: average RMSE in $\log_{10}$ scale of the state over $T_{PF} = 100$ time steps and for $N = 2^9$ particles.

NUTS can handle this type of model too.

All models described so far in this section are toy problems. The next section focuses on a highly dimensional real-world model.

6.4.2 Curse of Dimensionality

When it comes to using SIR PFs (equivalent to FL-SMC with $l = 0$), it is known that dimensionality may affect the estimation accuracy, a problem named curse of dimensionality which is shared with SMC samplers, as shown in Sections 6.2. One of the most common reasons is related to the number of sensors M_y and most importantly their standard deviation: if the sensors have very small variance, it is harder to effectively weight the particles, as RW proposals cancel the prior term in the importance weight equation. Another reasons could be the state transition system of equations, which may require a numerical integrator whose stability is influenced by the number of equations M.

This experiment is designed to test the performance of FL-NUTS and FL-HMC vs FL-SMC for models having increasing M and M_y. For this experiment the VAR model described in Section 4.4.5.3 is chosen, since it has significantly higher dimensionality than the models in the previous section. In this experiment, $N = 2^{17}$ particles are used, since it is the minimum N for convergence on this model when the dynamics is used as proposal, as shown in [52, 53].

The Streaming-Stan code for this model is found in Section C.2.5.4.

Table 6.4 illustrates the average RMSE in $\log_{10}$ scale of the state estimation for this experiment. All proposal distributions guarantee at least an acceptable estimate. However, it is also possible to observe better performance for FL-HMC and FL-NUTS by up to an order of magnitude in comparison with FL-SMC, which is struggling more due to the curse of dimensionality. In other words, using FL-HMC or FL-NUTS on such highly dimensional models provides better improvements than it does on single dimensional models, such as the flood water level one used in the previous section (see Table 6.2).

Symbol [unit]	**FL-SMC**	**FL-HMC**	**FL-NUTS**
Δ [cm]	-0.424	-0.988	-0.9817
G [cm]	-0.303	-1.394	-1.363
X_{ram} [cm]	-6.661	-7.681	-7.662
M_e [kg]	1.736	0.713	0.712
μ	-5.259	-5.778	-5.828
S_C [cm]	-4.489	-4.986	-4.954
S_M [cm]	-1.971	-2.677	-2.652
p_{he} [Torr]	-6.972	-7.944	-7.962
I [A]	0.334	-1.269	-1.286

Table 6.4: Vacuum Arc Remelting: average RMSE in $\log_{10}$ scale of each state quantity over $T_{PF} = 100$ time steps, for $N = 2^{17}$ particles and different proposals.

6.4.3 Long-Term Memory Models

As explained in Section 2.1.4, PFs work well as long as the measurements and the sampling technique are accurate. Fixed-Lag SMC methods offer the possibility to store more than one measurement and trajectories of more than one particle at each time step. This becomes useful in those scenarios where measurements are time-delayed or the process noise covariance is small which may affect the state estimate of a single iteration. In either of these situations, it is recommended to use $l > 0$ measurements at each iteration. This section investigates the performance of FL-SMC, FL-HMC and FL-NUTS on a model for which $l > 0$ may be beneficial.

To generate this scenario, the bearing-only tracking model is used with a much smaller process noise covariance, e.g. by setting $q = 0.005 \ll 5$. The measurement covariance is also considerably decreased to $10^{-4} \ll 0.33$ to favour HMC/NUTS moves over RW. After that, the same commands from the previous section can be used, apart from changes on the file name input arguments and using different values for l.

As we can see in Table 6.5, the accuracy of the estimates (again represented as average RMSE in $\log_{10}$ scale) for this model tend to converge to acceptable levels only when $l > 0$. In this case, it is also possible to observe that FL-HMC and FL-NUTS provide better accuracy than FL-SMC, due to the much smaller standard deviation of the sensor noise than the one in Section 6.4.1.3.

Symbol **[unit]**	**FL-SMC** $l = 0$	**FL-HMC** $l = 0$	**FL-NUTS** $l = 0$	**FL-SMC** $l = 2$	**FL-HMC** $l = 2$	**FL-NUTS** $l = 2$
(p_x, p_y) [m]	1.7574	1.6927	1.6853	−0.6439	−0.9573	−1.0245
(v_x, v_y) [m/s]	0.9658	0.9146	0.9482	−0.2468	−0.5563	−0.6278

TABLE 6.5: Bearing-Only Tracking: average RMSE in $\log_{10}$ scale of each state quantity over $T_{PF} = 100$ time steps, for $N = 2^9$ particles, different proposals and values of l.

6.4.4 Run-Time

The better accuracy that FL-HMC and FL-NUTS may provide over FL-SMC comes at a price. Using Leapfrog to propose new particles is more computationally intensive than making random moves. This section reports the average run-time of a single time step using increasing N for the three proposal distributions on the flood water level model. Other models are not taken into account for brevity, since the results for this experiment would be similar.

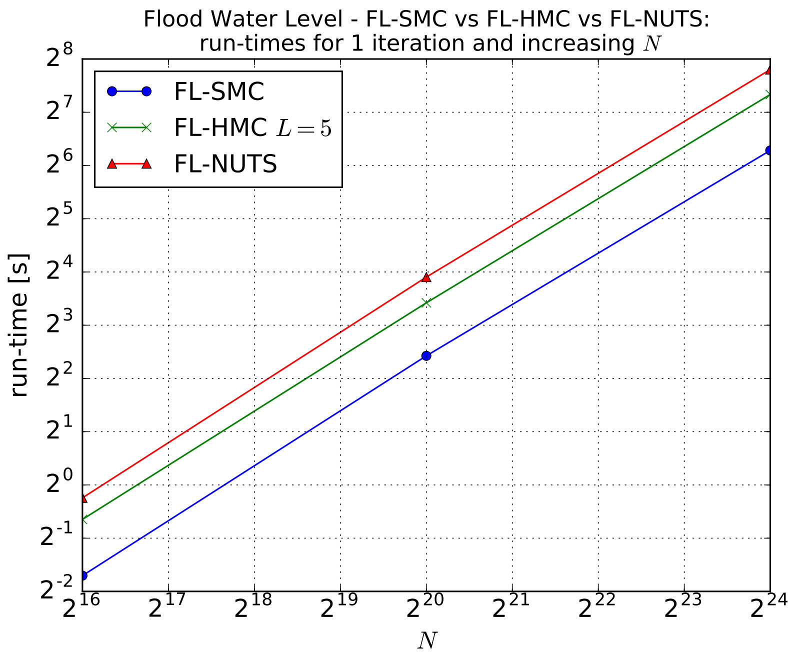


FIGURE 6.3: Flood Water Level - FL-SMC vs FL-HMC vs FL-NUTS: run-times of one time step for increasing N

As we can see in Figure 6.3, a FL-HMC method performing $L = 5$ Leapfrog steps and a FL-NUTS method may be two to three times slower than FL-SMC. In order to compensate for this side effect, it is then recommended to use parallel computing, especially the MPI+OpenMP RoSS redistribute designed in Chapters 4 and 5, which it has been proven to allow the PF to maintain a more linear speed-up than the same using other fully-balanced redistributions, such as B-R or N-R.

6.5 Conclusions

This chapter has presented SMC-Stan and Streaming-Stan, two extension packages for Stan which runs the MPI+OpenMP SMC methods described in the previous chapters, and use the same syntax of regular Stan to describe the model. More precisely, SMC-Stan embodies the MPI+OpenMP SMC sampler, such that it work on static models, as regular Stan does. Streaming-Stan is equipped with the MPI+OpenMP PF, such that it can solve dynamic models, which is something that regular Stan is currently unable to do. Each of these extensions of Stan have been equipped with three proposal distributions, which can be selected from command line, and differ for the type of MCMC moves to propose new samples: RW, HMC or NUTS. Moreover, FL-HMC and FL-NUTS, two of the three proposals in Streaming-Stan, are novel and employ HMC and NUTS on a FL-SMC method. These proposals provide up to an order of magnitude accuracy improvement vs currently existing FL-SMC methods, but they increase the run-time by at least a factor of two. Therefore, the reader and potential user is recommended to combine these novel proposals with the parallelisation algorithms described in the previous chapters, in order to compensate for this side effect.

Several improvements and exciting future work are ahead. The first one should be to investigate the benefits of using SMC^2, an alternative to PFs where each particle is itself the final estimate of an SMC sampler [20]. Another improvement avenue should be to edit Stan's back end to support GPU offload: at the moment, important Stan methods such as `log_prob`, `write_array` and `log_prob_grad` do not offer this possibility, because of the way the class `var` in Stan_math library has been designed. This issue is also linked to the results of Chapter 5, which underlines that further run-time improvements should focus on the performance of the IS step. A final idea to consider is to embed Streaming-Stan and SMC-Stan into existing popular libraries for Bayesian inference such as Stone Soup [46] and Scikit-learn [14, 71].

Chapter 7

Conclusions

This thesis has presented RoSS, a novel fully-balanced parallelisation algorithm for redistribute, the run-time bottleneck of SMC methods. The achieved time complexity is $O(\log N)$ (the same as the other tasks of SMC methods) on a hybrid distributed-shared memory architecture having $P \times T = N$ parallel cores. The code has been written in C++, using MPI to parallelise the tasks across P distributed computing nodes, and using OpenMP to achieve shared-memory parallelism within each node having T cores. A patent application for RoSS is currently filed.

The baseline for comparison is B-R, a state-of-the-art fully-balanced redistribute which achieves $O((\log N)^2)$ time complexity and which was originally presented on MapReduce. Therefore, in this thesis, B-R has first been ported from MapReduce to MPI, in order to compare the two algorithms on a common ground. The experiments have been run on several models, some describing toy problems and others representing highly dimensional real-world problems. The research findings emphasise that an SMC method using B-R becomes almost an order of magnitude faster on a cluster of 256 cores when switching to RoSS, and achieves up to a 160 speed-up factor vs a single-core SMC method. For the same DOP, B-R accounts for up to 85% of the total run-time of the SMC method, while it is interesting to see that RoSS has not yet emerged as bottleneck over IS, another component of SMC methods which is tasked with sampling and weighting the N particles at each SMC iteration.

In order to make the improvements widely available to industries and research community, this thesis presents Streaming-Stan and SMC-Stan, two extension packages for Stan which allow the user to describe respectively dynamic and static models using the same syntax used by Stan, but solve them with an SMC method (in the form of a FL-SMC method and an SMC sampler respectively) running on a supercomputer. Since the syntax and compiler of Stan have been preserved, Streaming-Stan and SMC-Stan can easily interface with other popular languages, such as MATLAB, Python, R and Julia. Also, the user is still able to run NUTS MCMC method in SMC-Stan if they choose to, as in regular Stan. The installed SMC sampler and FL-SMC method both equip all parallel MPI+OpenMP algorithms implemented in this thesis, including RoSS, along with three optional proposal distributions to sample and weight the particles. In particular, FL-HMC and FL-NUTS, two of the three proposals of Streaming-Stan, are novel and provide up to an order of magnitude accuracy improvement vs regular FL-SMC methods. The results are again collected on several models, representing toy problems or highly-dimensional real-world problems.

The findings are encouraging but several advances can still be made. Since that the run-time results have shown that IS is likely to be slower than RoSS, the focus should now be on improving the performance of IS, independently of the model. Streaming-Stan and SMC-Stan currently run on a cluster of CPU cores: therefore, one way to achieve this goal is to redesign Stan's back end to support GPU offload, in order to exploit the extra speed-up that graphics cards usually provide over CPUs. Another consideration to make is that MVR is the preferred resampling scheme for this thesis, since it is used in several referenced work. Then, future work should also be focused on comparisons with different schemes in terms of accuracy and run-time, to also investigate if other schemes could lead to further simplification of the parallel redistribution presented in this thesis. Apart from ideas to improve accuracy and run-time performance, the focus should also be on designing a user-friendly interface to call Streaming-Stan and SMC-Stan from widely popular Bayesian inference libraries, such as Stone Soup or Scikit-learn.

Appendix A

Distributed, Shared and Hybrid Memory Architectures

This appendix provides general information about distributed, shared and hybrid memory architectures along with a brief tutorial about MPI and OpenMP, two of the most popular APIs for these environments. Further details can be found in [66, 87]

A.1 Distributed Memory Architectures and MPI

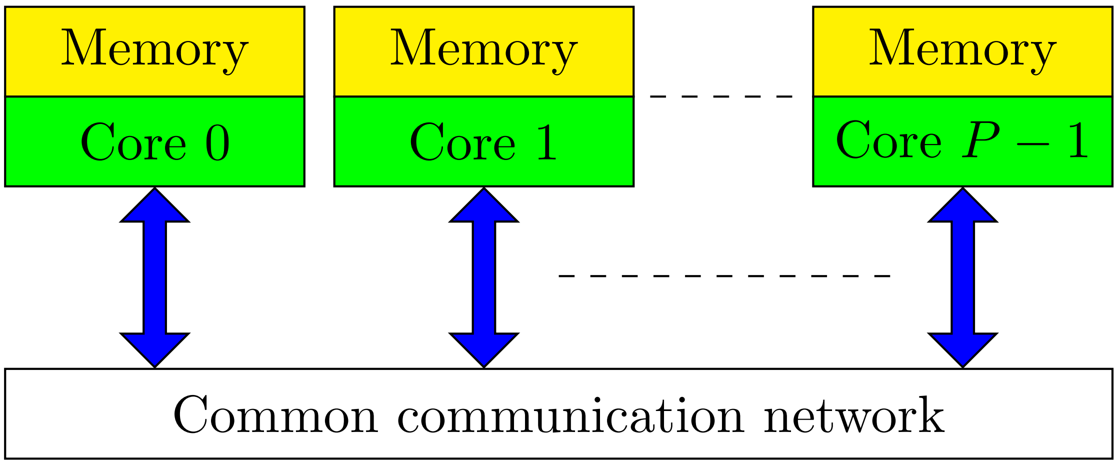


Figure A.1: Distributed Memory Architecture

DMAs are a type of parallel system which are inherently opposed to SMAs. In this environment, the memory is distributed over the cores and each core can only directly access its own private memory. Exchange of information stored in the memory of the other cores is achieved by sending/receiving explicit messages through a common communication network.

DMAs provide several advantages over SMAs such as scalable and larger DOP, scalable and larger memory which can also be accessed by its proprietary core with no interference. The main disadvantage is the cost of communication and the consequent data movement. This may affect the speed-up relative to a single core version of the same algorithm.

There are lots of APIs for DMAs and all of them are suitable for this thesis. MPI is arguably the most popular API for DMAs due to its intuitive syntax. In this API, we have P MPI processes which are created and mapped to the physical cores by using `MPI_Init()`. The typical approach to maximise the performance is to map one MPI process to each core. Then the MPI processes are uniquely identified by a rank $p = 0, 1, \ldots, P-1$: the rank is assigned by calling `MPI_Comm_rank()`. Once created and mapped, the MPI cores need to be connected. This is achieved by using `MPI_Comm_size()` which takes the communicator variable, the list and number of MPI ranks that want to register to that communicator. Each communicator can also be split into two or more new communicators by using `MPI_Split()` which takes in input the father communicator, the list of child communicators and the lists of ranks that want to use the child communicators.

The registered ranks within the same communicator can use explicit send/receive communication routines to exchange messages. Therefore, these routines require knowledge of the MPI ranks involved on both ends: sending and receiving. There exists several MPI communication routines, which are classified depending on the communication topology (see Figure A.2). The following table provides a list of common routines that have been used for this thesis.

Name	**Topology**	**Description**	**Time Complexity**	**Space Complexity**
`MPI_Alltoall`	Flat tree	All ranks send and receive a message	$O(P \log_2 P)$	$O(P)$
`MPI_Gather`	Flat tree	A rank sends a different message to the other ranks	$O(P)$	$O(P)$
`MPI_Scatter`	Flat tree	Multiple ranks sends a different message to one rank	$O(P)$	$O(P)$
`MPI_Send`	One-to-one	A rank sends a message to another rank	$O(1)$	$O(1)$
`MPI_Recv`	One-to-one	A rank receives a message from another rank	$O(1)$	$O(1)$
`MPI_Sendrecv`	One-to-one	Two ranks exchange a message	$O(1)$	$O(1)$
`MPI_Bcast`	Binary tree	A rank broadcasts the same message to the others	$O(\log_2 P)$	$O(1)$
`MPI_Reduce`	Binary tree	It computes reduction. A root rank holds the result.	$O(\log_2 P)$	$O(1)$
`MPI_Allreduce`	Binary tree	It computes reduction. All ranks hold the result.	$O(\log_2 P)$	$O(1)$
`MPI_Scan`	Binary tree	It computes inclusive Cumulative Sum.	$O(\log_2 P)$	$O(1)$
`MPI_Exscan`	Binary tree	It computes exclusive Cumulative Sum.	$O(\log_2 P)$	$O(1)$

Table A.1: Common MPI communicators.

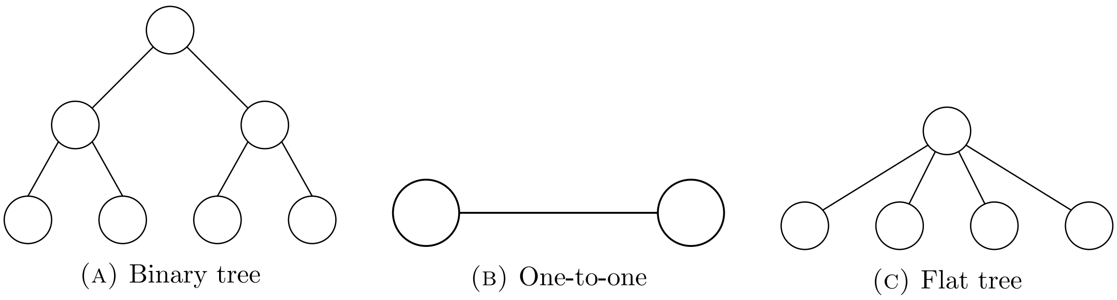

(A) Binary tree (B) One-to-one (C) Flat tree

FIGURE A.2: MPI node topologies

A.2 Shared Memory Architectures and OpenMP

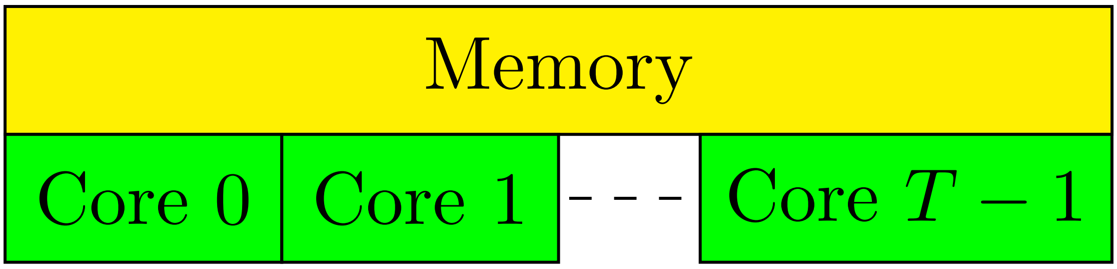


FIGURE A.3: Shared Memory Architecture

SMAs are a type of parallel system which are fundamentally different to DMAs. In these architectures, the cores can simultaneously access the same memory which can be used to share information between the threads.

The main advantage over DMAs is therefore fast communication between the T cores since SMAs achieve this by accessing the system-wide shared memory. However, SMAs have two main disadvantages relative to DMAs: memory access increases with T and the largest systems that use DMAs are bigger than the largest that use SMAs alone.

As said for DMAs, any shared memory API is suitable for this thesis. OpenMP is chosen in this case for its simple and intuitive syntax which makes it one of the most popular programming models for SMAs. OpenMP applies the fork-join model to set up T parallel threads which are uniquely identified by an $id \in \mathbb{Z}$. Once created, the threads can concurrently execute bodies of instructions which are coded within directives for the compiler called pragmas. Every pragma directive is identified by the following syntax: `#pragma omp clause1(arg list) clause2(arg list) ... clauseN(arg list)` where the various clauses are chosen based on the type of parallelism which is needed for the related body of instructions. The following is a list of the most important clauses to understand the concepts explained in Chapter 5 of this thesis:

- `parallel`: it takes no arguments and activates multithreading for the related body of instructions;
- `for`: it is used after `parallel`, takes no arguments and activates multithreading over a for loop;

- `schedule`: it is used after `parallel for` and establishes the partitioning strategy of the iteration space. Typical arguments are `static` or `runtime`. If `static` the iteration space gets distributed equally. If `runtime` the fastest threads run more iterations;
- `num_threads`: it takes either an integer number or an integer variable which correspond to the number of threads to create;
- `private`: it takes a list of previously declared variables of which every parallel thread create a new private copy, initialised to a random value;
- `firstprivate`: it takes a list of previously declared variables of which every thread create a new private copy, initialised to the value of the variable before the pragma directive;
- `simd`: it takes no arguments and activates vector parallelism;
- `reduction`: it takes in input the type of reduction operation to compute, followed by the name of the variable to reduce. The syntax for sum is `+:variable_name`, while for max is `max:variable_name`.

A.3 Hybrid Memory Architectures Using MPI and OpenMP

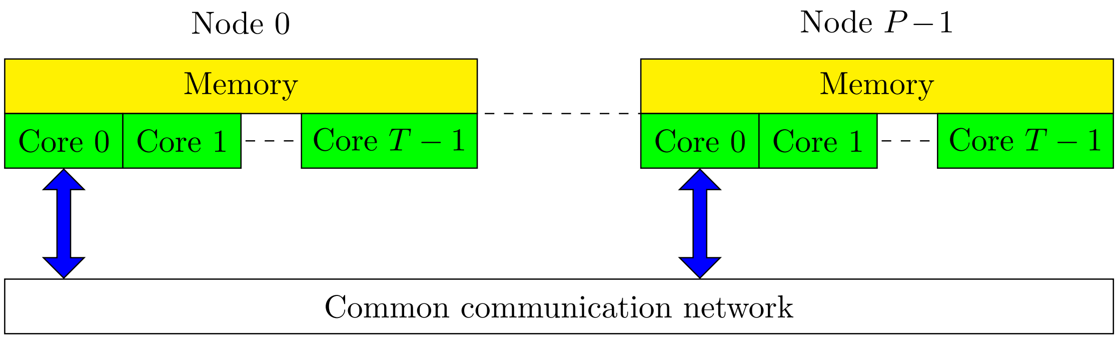


FIGURE A.4: Hybrid Memory Architecture

Hybrid memory architectures consist of multiple computing nodes connected as in a DMA, where each of these nodes is itself a SMA. These architectures combine the benefits of DMAs and SMAs, i.e. scalable memory with the number of nodes, and fast data-sharing within each node. The main disadvantage is a considerable increased coding complexity, as some single node task may be hard to parallelise efficiently on SMAs.

One of the most common approaches to implement hybrid DMA-SMA code is to extend existing MPI codes with OpenMP. This is done by calling `MPI_init_thread()` in place of `MPI_init()` and using the proper pragma directives on the top of parallelisable single node bodies of instructions. `MPI_init_thread()` takes in input a flag variable which sets up the level of thread support. The possible flag values are:

- `MPI_THREAD_SINGLE` if only the master thread is allowed to call MPI routines outside a parallel region;
- `MPI_THREAD_FUNNELED` if only the master thread is allowed to call MPI routines inside a parallel region;
- `MPI_THREAD_SERIALIZED` if all threads are allowed to call MPI routines inside but only one at the time;
- `MPI_THREAD_MULTIPLE` if all threads are allowed to concurrently call MPI routines.

Appendix B

Stan

Stan is a probabilistic programming language for statistical modeling, data analysis, and prediction making on a static model. It is currently used by over ten thousand users. Its popularity is mostly due to the following reasons:

- its syntax is simple, intuitive and very succinct: in a few code lines it is possible to design complex models that may take hundreds of lines in other languages, such as C++;
- it provides a large library of statistical functions;
- the back end is written in C++, which makes the overall run-time highly performing;
- it can interface with many popular programming languages such as MATLAB, Python, R and Julia;
- it uses NUTS to draw samples, a highly accurate gradient-based proposal distribution which is explained in Section 2.2.3;
- in order to compute the gradient, Stan provides a highly performing gradient calculator based on auto-differentiation.

This appendix provides a brief description of the most important features of Stan and its back end, in order to follow the concepts explained in Chapter 6. The reader is referred to Stan user's guide [84] for extra details.

B.1 How to Use CmdStan

Since the final goal of this thesis is to extend Stan's back end to use other Bayesian inference methods such as SMC samplers and PFs, CmdStan, the command line version of Stan, is used as this is the only alternative which is not protected by licenses.

Once CmdStan is correctly built, users must write a .stan file which describes the model. This file has to be compiled by `stanc`, the compiler for Stan. If the compilation is successful, a new .hpp file is generated, containing a C++ class describing the statistical model in the original .stan file. After that, the .hpp file needs to be compiled with rest of the back end. The default compiler for this step is `g++` but others are available, such as `mpic++`, upon previous installation of compatible MPI libraries, such as OpenMPI. After this second compilation, a new executable file is generated and can be run from command line using the `./` command. Several arguments need to be provided, depending on the desired output, such as the number of samples to draw. The followings are the most important arguments to be given from command line:

- `method=` which defines the type of problem to be solved. Since sampling is the main interest in this work, the typical value to give is `sample` and is the only one considered in this thesis;
- `algorithm=` which is used to define the sampling method. In CmdStan the two accepted values are `hmc` for NUTS and `fixed_params` which is not covered in this thesis;
- `num_samples=` which establishes the number of samples to generate;
- `num_warmup=` which establishes the number of samples to generate during the burn-in phase;
- `data file=` which provides the name of the file containing the data, if there is any. Supported formats are .R, .json and .csv;
- `output file=` an optional argument to provide the name of the output file containing the generated samples. If not given, the default output file name is output.csv.

B.2 CmdStan: Back End Summary

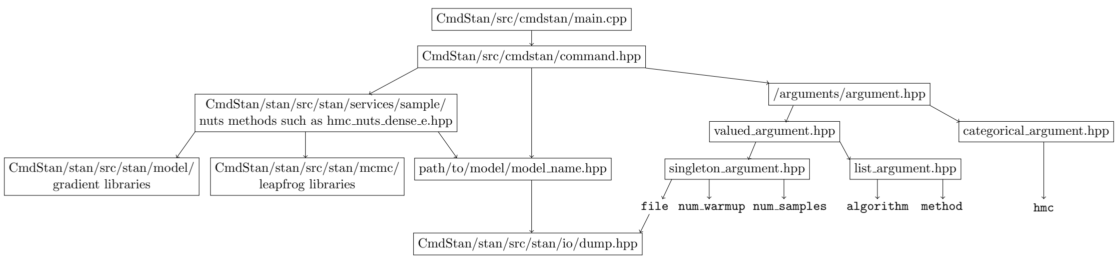

CmdStan/src/cmdstan/main.cpp
CmdStan/src/cmdstan/command.hpp
CmdStan/stan/src/stan/services/sample/ nuts methods such as hmc_nuts_dense_e.hpp
/arguments/argument.hpp
valued_argument.hpp
categorical_argument.hpp
CmdStan/stan/src/stan/model/ gradient libraries
CmdStan/stan/src/stan/mcmc/ leapfrog libraries
path/to/model/model_name.hpp
singleton_argument.hpp
list_argument.hpp
file
num_warmup
num_samples
algorithm
method
hmc
CmdStan/stan/src/stan/io/dump.hpp

FIGURE B.1: Stan - back end

The graph in Figure B.1 briefly summarises the most important classes and libraries in Stan's back end with a view to easily and efficiently explaining which folders need to be created or extended to install an SMC sampler and a PF in Stan. As we can see, the main includes the library command.hpp from which all argument classes and sampling method classes are accessible. More precisely, the only method in command.hpp is tasked with three assignments: parse the input arguments, generate an object of the model class, call the selected method by feeding it the model object and the required received arguments. In doing so, three types of argument classes (child to the argument main class) are necessary:

- `list_argument` class which is used to parse the values of `method` and `algorithm` from command line;
- `singleton_argument` class which is used to parse any integer, float or string argument such as `num_samples`, `num_warmup` and `data file`;
- `categorical_argument` class which is used to parse the possible values of other list arguments, such as different variants of NUTS which is given to `algorithm`.

The model class is usually found in an arbitrary folder and is referred to on any library or class in Stan's back end by using the alias `stan_model`. The constructor of the model class requires an object of the dump class, called `data_var_context` in command.hpp, which consists of a C++ map containing names and values of the data file. In order to access the data file, `data_var_context` requires the file path which is found in the string argument for `data file`. The model constructor (whose body of instructions is changed at compile time by `stanc` depending on the content of the .stan model) accesses the data values by name and saves them as private members.

The model object and the other relevant arguments are fed to the selected algorithm, whose calling methods are found in the CmdStan/stan/src/stan/services/ folder.

B.3 Syntax

Models are described in .stan file which is composed of up to seven blocks, each one defining a part of the model. A brief description of each block follows:

- **data** (optional but typically used): it declares constants (e.g. physical constants or array dimensions) and other data, such as measurements. Possible keywords for data types are `int` for integers, `real` for floating point numbers, and `vector` or `matrix` for structures. Arrays of `int` and `real` can also be declared by adding squared brackets containing the dimension of the array; the same thing has to be done for `vector` and `matrix`. The values are stored in a data file whose name must then be given in input from command line;
- **transformed data** (optional): it is used to declare new data which is defined as function of the data in the **data** block. The keywords for the data types are the same as in the **data** block;
- **parameters**: it declares the parameters to sample directly. The keywords for the data types are the same as in the **data** block;

- **transformed parameters** (optional): it declares and defines the parameters that are not sampled directly but computed as function of the **parameters** during the sampling iterations. The keywords for the data types are the same as in the **data** block;
- **model**: it defines the posterior distribution in logarithmic scale. The keyword for the posterior is `target` and every factor which contributes to the posterior may be added to `target` by using the following syntax: `target +=`;
- **generated quantities** (optional): it declares and defines output parameters which are not sampled directly but computed as function of the **parameters** and the **transformed parameters** after the sampling iterations. The keywords for the data types are the same as in the **data** block;
- **functions** (optional): it declares and defines functions which may be useful in the rest of the Stan file, e.g. to define **transformed data**, **transformed parameters** or statistical functions to use in the **model** and that are not built-in.

B.3.1 Example: Student-t Distribution

This example describes how to use Stan to sample from a student-t distribution which has the following multivariate form:

$$\begin{aligned} p(\mathbf{X}|\mathbf{Y}) = \Gamma\left(\frac{\nu+M}{2}\right) - \Gamma\left(\frac{\nu}{2}\right)\nu^{0.5\cdot M}p^{0.5\cdot M}|\mathbf{\Sigma}|^{0.5}+ \\ +\left(1+\frac{1}{\nu}(\mathbf{X}-\mu)^{\intercal}|\mathbf{\Sigma}|^{-1}(\mathbf{X}-\mu)\right)^{-\frac{\nu+M}{2}} \end{aligned} \tag{B.1}$$

where the model data is simply the input dimension M, the degrees of freedom ν, the mean μ, and the variance $\mathbf{\Sigma}$. Here, $M = 1$, $\nu = 7$, $\mu = 0$ and $\mathbf{\Sigma} = 1$. A suitable .stan file follows:

```
data {
   real nu; //degrees of freedom
   real mu; //mean
   real sigma; //covariance
}
parameters {
   real x;
}
model {
   target += student_t_lpdf(x | nu, mu, sigma);
}
```

where the data can be stored in the following .R file:

```
nu <- 7
mu <- 0
sigma <- 1
```

If the .R file is named, for example, student_t.data.R and the executable file is named, for example, student_t, after compilation the command to run is:

```
$ ./student_t method=sample algorithm=hmc num_samples=128 \
   num_warmpup=100 data file=student_t.data.R
```

which generates 128 estimates of the mean value, after performing 100 burn-in iterations.

B.3.2 Example: Neal's Funnel

The Neal's funnel described in Section 2.2.3.1, whose target is defined in Equation (2.39), can be describe by the following .stan file, called neals_funnel.stan, which can also be found in Stan user's guide [84]:

```
parameters {
   real y_raw;
   vector[9] x_raw;
}
transformed parameters {
   real y;
   vector[9] x;
   y = 3.0 * y_raw;
   x = exp(y/2) * x_raw;
}
model {
   y_raw ~ std_normal(); // equals target += normal_lpdf(y_raw|0, 1)
   x_raw ~ std_normal(); // equals target += normal_lpdf(x_raw|0, 1)
}
```

After compilation, an arbitrary number of samples, e.g. $N = 1024$, can be generated in regular Stan by running the following command:

```
$ ./neals_funnel method=sample algorithm=hmc num_samples=1024 \
   num_warmpup=100.
```

Appendix C

How to Install SMC Methods in Stan

This appendix describes how to set up SMC-Stan and Streaming-Stan by respectively installing an SMC sampler and a PF in Stan. Although this thesis is mostly focused on PFs, it is more intuitive to explain how to set up SMC-Stan first. This is because the changes to be made in the back end are fewer for SMC-Stan and mostly have to be repeated for Streaming-Stan. The reader is referred to Appendix B for details about Stan's back end.

C.1 How to Set up SMC-Stan

The first thing to do is create a new folder, here called smc, containing the SMC sampler libraries which consists of the parallel algorithms described in Chapters 3, 4 and 5 and Algorithm 3. This folder should be created in the following path: CmdStan/stan/src/stan/ where all sampling methods, such as NUTS, already are.

Once the SMC sampler has been copied, it is necessary to create any extra input argument that Algorithm 3 requires and is not already available in regular Stan. The argument `num_samples` can be used for the number of particles N per SMC iteration. One argument that should be created is, for example, `Tsmc`, which allows the user to input T_{SMC} for Algorithm 3. Therefore, a new argument class, called for example `arg_Tsmc`, has to be defined as child to the `singleton_argument` class, since it is an integer input argument. A possible C++ code follows:

```
#ifndef CMDSTAN_ARGUMENTS_ARG_TSMC_HPP
#define CMDSTAN_ARGUMENTS_ARG_TSMC_HPP

#include <cmdstan/arguments/singleton_argument.hpp>

namespace cmdstan {
   class arg_Tsmc: public int_argument {
   public:
      arg_Tsmc(): int_argument() {
         _name = "Tsmc";
         _description = "Total number of SMC iterations";
         _validity = "1 < Tsmc";
         _default = "100";
         _default_value = 100;
         _constrained = true;
         _good_value = 2.0;
         _bad_value = -1.0;
         _value = _default_value;
      }

      bool is_valid(unsigned int value) { return value > 1; }
   };
}
#endif
```

Any other valued input argument to Algorithm 3, such as one for the number of OpenMP threads T, has to be created the same way. The C++ code is omitted for brevity.

Now it is necessary to enable the user to invoke Algorithm 3 from command line, the same way regular Stan does with NUTS. This is achieved by the following four steps:

1. A new possible value for argument `algorithm` has to be registered. This first requires a new argument class, called for example `arg_smcs`, child to the existing class `categorical_argument`. A possible C++ code is:

   ```
   #ifndef CMDSTAN_ARGUMENTS_ARG_SMCS_HPP
   #define CMDSTAN_ARGUMENTS_ARG_SMCS_HPP

   #include <cmdstan/arguments/categorical_argument.hpp>
   namespace cmdstan {
      class arg_smcs: public categorical_argument {
      public:
         arg_smcs() {
            _name = "smcs";
            _description = "SMC sampler";
            _subarguments.push_back(new arg_Tsmc());
            _subarguments.push_back(new arg_T());
            _subarguments.push_back(new arg_proposal());
         }
      };
   }
   #endif
   ```

   The role of the argument `arg_proposal` will be explained in the following sections.

2. Now, `arg_smcs` needs to be registered in the possible values that the command line argument `algorithm` accepts. This is done by adding `arg_smcs()` as input member in the constructor of the already existing `arg_sample_algo` class as follows:

```
#ifndef CMDSTAN_ARGUMENTS_ARG_SAMPLE_ALGO_HPP
#define CMDSTAN_ARGUMENTS_ARG_SAMPLE_ALGO_HPP

#include <cmdstan/arguments/list_argument.hpp>
#include <cmdstan/arguments/arg_hmc.hpp>
#include <cmdstan/arguments/arg_fixed_param.hpp>
#include <cmdstan/arguments/arg_smcs.hpp>

namespace cmdstan {
   class arg_sample_algo: public list_argument {
   public:
      arg_sample_algo() {
         _name = "algorithm";
         _description = "Sampling algorithm";

         _values.push_back(new arg_hmc());
         _values.push_back(new arg_fixed_param());
         _values.push_back(new arg_smcs());

         _default_cursor = 0;
         _cursor = _default_cursor;
      }
   };
}
#endif
```

3. In order to follow the structure of Stan's back end, a new method, called for example `smc_sampler`, has to be created in the folder CmdStan/stan/src/stan/services/sample/. The role of this method is simply to print out the relevant input arguments to the output .csv file, before calling Algorithm 3, which is found in the CmdStan/stan/src/stan/smc folder, as previously mentioned in this section;

4. In command.hpp an object of `arg_sample_algo` called `algo` is created. The value given to the constructor of `algo` is checked by using a sequence of nested if-else branches and, depending on the given value, a method from CmdStan/stan/src/stan/services/sample/ is invoked. Therefore, the next step is to create another `else if` branch which checks whether the given value is equal to `smcs`. If true, `smc_sampler` is invoked after reading its required input arguments, such as `num_samples`, `Tsmc` and `T`. This passage is relatively straightforward and the reader is simply referred to repository of CmdStan https://github.com/stan-dev/cmdstan to understand how to proceed.

C.1.1 Proposal Distributions

As anticipated elsewhere in this thesis, the goal is to install an SMC sampler which provides multiple proposal distributions (apart from regular RW moves) by employing better MCMC samplers than RW. To achieve that, it is necessary to first install in the argument folder a new class called `arg_proposal`, child to the `list_argument` class. The C++ code is omitted for brevity but the reader can refer to the `arg_sample_algo` class defined in the previous section. This creates a new command line input argument called `proposal` which the user can use to select the preferred proposal to be used by the IS step in Algorithm 3. The back end is programmed to reject the command line if `proposal` is defined but `algorithm` is not given `smcs` as value.

A new class for each type of proposal distribution needs to be defined, in order to become a new value that the argument `proposal` accepts. The names of these classes will have to be registered in the `_values` field of `arg_proposal`.

C.1.1.1 Random Walk

For RW moves, a class called `arg_random_walk_proposal` has to be defined. This class will have to be child to the `categorical_argument` class and have `_name` field equal to the keyword `rw`. For extra implementation details the reader can consult the C++ for the next proposal defined in the next section. Therefore, if `rw` is given to `proposal` from command line, IS in Algorithm 3 will sample each new particle using (2.22) and weight the particles as in (2.23).

C.1.1.2 HMC

To use HMC as proposal, it is necessary to register a possible value for the new argument `proposal`, called for example `hmc`. Therefore, a class called `arg_hmc_proposal` has to be defined, whose C++ code can be the following:

```cpp
#ifndef CMDSTAN_ARGUMENTS_ARG_HMC_PROPOSAL_HPP
#define CMDSTAN_ARGUMENTS_ARG_HMC_PROPOSAL_HPP

#include <cmdstan/arguments/categorical_argument.hpp>

namespace cmdstan {
   class arg_hmc_proposal: public categorical_argument {
   public:
      arg_hmc_proposal() {
         _name = "hmc";
         _description = "HMC proposal distribution";
         _subarguments.push_back(new arg_stepsize());
         _subarguments.push_back(new arg_num_leapfrog_steps());
      }
   };
}
```

```
#endif
```

Also, two extra input arguments are required for this proposal: one for the number of leapfrog steps L, and one for the step size Δh. Therefore, two extra classes, called for example `arg_num_leapfrog_steps` and `arg_stepsize`, must be created and added to the member `_subarguments` of the constructor of `arg_hmc_proposal`. For these two arguments, `num_leapfrog_steps` and `stepsize` have been chosen as names. The C++ codes for `arg_num_leapfrog_steps` and `arg_stepsize` are omitted for brevity, since they simply consist of child classes to `singleton_argument`, one in the form of integer argument, the other in the form of real argument. In order to compute the gradient in Leapfrog, Stan provides a callable method named `log_prob_grad`, found in CmdStan/stan/src/stan/model, which is designed to perform a fast auto-differentiation on `log_prob`, the method which computes $p(\mathbf{X}|\mathbf{Y})$ in the model class. Hence, `log_prob_grad` is also used in the libraries in CmdStan/stan/src/stan/smc.

C.1.1.3 NUTS

The steps to include NUTS as proposal are the same as in the previous section. Therefore, the C++ code for `arg_NUTS_proposal` is omitted for brevity as it is equivalent to the one for `arg_hmc_proposal`, apart from the member `_name` being assigned to `nuts` as value, and without adding `arg_num_leapfrog_steps` to `_subarguments`, since NUTS uses Leapfrog with an adaptive number of steps.

C.2 How to Set up Streaming-Stan

The first thing to be done is to create a .hpp file defining Algorithm 4. As mentioned in Chapter 2, a PF can be designed as an FL-SMC method using one measurement at each time step. Indeed, Equation (2.28) can be manipulated to be equal to (2.13) when the fixed lag $l = 0$. This is why a single .hpp file is sufficient for both PFs and FL-SMC methods. Also, as mention in Chapter 2, these two SMC methods only differ from SMC samplers for the importance weight equation, but require for each task the same parallelisation methods, described in Chapters 3, 4 and 5. Therefore, the .hpp file for Algorithm 4 can simply replace the .hpp file for Algorithm 3 in CmdStan/stan/src/stan/smc. After that, the same steps described in Section 6.2 to register a new valid value for the argument `algorithm` must be repeated to register `pf` as well. These steps are summarised as follows:

- create a new argument class, called for example `arg_pf`, child to the existing categorical argument class. The code should be identical to `arg_smcs`, but having the member `_name` equal to the value `pf`. Also, to be consistent with the notation of this thesis, the argument class `arg_Tsmc()` has been substituted with an identical argument class called `arg_Tpf()` which registers `Tpf` as new possible input argument in Streaming-Stan, in order to input the number of time steps T_{PF};

- extend the constructor of `arg_sample_algo` by adding a new element to the member `_values`, precisely `arg_pf();`
- create a new .hpp file in CmdStan/stan/src/stan/services/sample containing a method, called for example `particle_filter`, which is tasked with first printing out to the output .csv file a summary of the input arguments list, before calling Algorithm 4;
- add another else-if branch in command.hpp to check whether `algo->value()` is equal to `pf` and, if true, call the method `particle_filter` after creating an object for each input argument, as done for the method `smc_sampler`;

At this point, running a PF in Stan would require:

1. the possibility to re-read the data file without recompiling the model, in order to emulate a real-time income of measurement $\mathbf{Y}_t$;
2. the possibility to describe the current state $\mathbf{X}_t$ in the **parameters** or **transformed parameters** block and save it at run-time to the old state $\mathbf{X}_{t-1}$ which should, however, be defined in the **data** block.
3. the possibility to describe both target and initial distribution and compile them into the same executable file.

At the moment, none of them are possible in Stan. The first two are currently impossible because the data variables become private members of the C++ model class generated by the compiler `stanc`. Therefore, the model class does not provide get-set methods to modify the data at run-time. The last constraint requires description and compilation of two models at the same time. The following three sections describe a possible solution for each of these limitations.

C.2.1 Real-Time Measurement

To overcome the first limitation without making changes to the compiler, it is necessary to manipulate to the input given to the constructor of the model class, instead of trying to change the content of the model object once this has been created. As previously mentioned in Section B.2, the constructor of the model class requires `data_var_context`, an object of the `dump` class in CmdStan/stan/src/stan/io, which has a C++ map containing the content of the data file. This class already provides get methods to return the value of a specific integer or real member of the map, if its name is known. Therefore, since `dump` is hard-coded and is not edited at run-time by `stanc`, it is only needed to add set methods to search by name for a value in the map that we want to update. This means that now the data file can be re-read, and then `data_var_context` can be edited and given to the constructor of the model class, in order to create a new model object. To achieve that, `data_var_context` and the path to the data file must be added to the input list of `particle_filter`, Algorithm 4 and its task `New_Measurement()`. At each time step t, the MPI node having rank $p = 0$ is tasked with reading the data file again, which is expected to be changed at run-time by an external measurement source, and saving its content to a new temporary object of `dump`. More precisely, the PF expects to read new values only for the subset of variables in the **data** block that are changed in real-time, e.g. $\mathbf{Y}_t$. By comparing this temporary object with `data_var_context`, it is possible to infer which data has changed, such that rank $p = 0$ can then broadcast the new content to the other MPI nodes by using `MPI_Bcast`. After that, each MPI node can independently update the content of `data_var_context`, create a new model object, and destroy the old one. It is now possible to update at run-time the data block of .stan in $O(\log P)$, the time complexity of `MPI_Bcast` (see Table A.1).

C.2.2 Old State Declared as Data

The second limitation is trickier than the first because, in order to achieve this goal, Streaming-Stan has to be able to access the value of $\mathbf{X}_t$ and $\mathbf{X}_{t-1}$, and then copy the content of $\mathbf{X}_t$ to $\mathbf{X}_{t-1}$. One naive solution would be to force the user to declare $\mathbf{X}_{t-1}$ and $\mathbf{X}_t$ in the **parameters** block and place them in a pre-agreed order, such that they can be managed within Algorithm 4. However, this solution would require complex changes to Stan's back end, such as the interface of `log_prob_grad` (which is designed to compute the gradient of the whole **parameters** block), besides adding constraints to Stan's syntax. On the other hand, this limitation becomes even harder to overcome if the constraints are once again to make no major changes to `stanc` or Stan's syntax. This is due to the following reasons:

- The most intuitive block to declare $\mathbf{X}_{t-1}$ in is **data**, while $\mathbf{X}_t$ is inherently part of the **parameters** or **transformed parameters**. This is because, at each time step t, $\mathbf{X}_t$ is to be sampled, while $\mathbf{X}_{t-1}$ is constant and a given term to the posterior. At the moment, however, there is no method in any class that copies parameters to data, as this feature is not contemplated in regular Stan;

- $\mathbf{X}_{t-1}$ and $\mathbf{X}_t$ are only subsets of **data** and **parameters** (since others may be needed and declared) and their variable names are arbitrary.

It is currently possible to access $\mathbf{X}_{t-1}$ by name from `data_var_context` thanks to the changes described in the previous section. When it comes to the parameters, and hence $\mathbf{X}_t$, it is already possible to access them by address. All parameters (i.e. **parameters**, **transformed parameters** and **generated quantities**) are indeed placed by `stanc` onto a C++ vector, in the same order they are declared. This compiler also generates two useful methods of the model class, called `get_param_names` and `get_dims` which respectively return name list and dimension list of all parameters in the same order they are stored to the vector. Therefore, what is currently missing is a mechanism for Streaming-Stan to know (before running the executable file) the names of the parameters and data variables that form $\mathbf{X}_t$ and $\mathbf{X}_{t-1}$, along with the order these variables are declared in, which may also be totally arbitrary.

The solution to this limitation consists of adding two new input arguments `old_state` and `state`. The values of these arguments are designed to be a list of strings separated by a comma, precisely the variable names for $\mathbf{X}_{t-1}$ and $\mathbf{X}_t$, in the order which the variable of $\mathbf{X}_t$ has to overwrite the variables of $\mathbf{X}_{t-1}$. For example, if a .stan file has $\mathbf{X}_t = \{\texttt{Alpha}, \texttt{Beta}\}$ and $\mathbf{X}_{t-1} = \{\texttt{Gamma}, \texttt{Delta}\}$, by giving the following input argument `state=Alpha,Beta` and `old_state=Gamma,Delta`, at each SMC iteration `Alpha` overwrites `Gamma` and `Beta` overwrites `Delta`. Broadly speaking, the value of $\mathbf{X}_t$ is read by address using the names in `state` as described above, and its value overwrites some variables in `data_var_context`, precisely those having the names in `old_state`. Then, the updated `data_var_context` is used to create a new model object. This operation must obviously be done for each particle during IS, but since IS is embarrassingly parallel, the overall overhead is negligible for a high DOP. Stan does not currently offer any argument class which takes in input a list of values linked to the same argument keyword. Also, the `singleton_argument` class is not easy to re-factor to suit these needs, since it is built to have a keyword per each value. Although re-factoring `singleton_argument` should be considered as future work, the current solution has been to create a new argument class, called `multiton_argument` and child to `valued_argument`, whose C++ code is omitted for brevity. The C++ codes for `old_state` and `state` are, however, provided below:

```
#ifndef CMDSTAN_ARGUMENTS_ARG_PF_OLD_STATE_HPP
#define CMDSTAN_ARGUMENTS_ARG_PF_OLD_STATE_HPP

#include <cmdstan/arguments/multiton_argument.hpp>
#include <vector>
#include <string>
#include <regex>

namespace cmdstan {
   class arg_old_state: public multi_string_argument {
   public:
      arg_old_state(): multi_string_argument() {
         _name = "old_state";
         _description = std::string("data subset for the old state in
    PF");
         _validity = "Names in data block separated by ,";
         _default = {};
         _default_value = {};
         _constrained = false;
         _value = _default_value;
      }
      bool is_valid(std::string value){
         return !std::regex_match(value, std::regex( (
    "((\\+|-)?[[:digit:]]+)(\\.(([[:digit:]]+)?))?" ) ) );
      }
   };
}
#endif
```

```
#ifndef CMDSTAN_ARGUMENTS_ARG_PF_STATE_HPP
#define CMDSTAN_ARGUMENTS_ARG_PF_STATE_HPP

#include <cmdstan/arguments/multiton_argument.hpp>
#include <vector>
#include <string>
#include <regex>

namespace cmdstan {
   class arg_state: public multi_string_argument {
   public:
      arg_state(): multi_string_argument() {
         _name = "state";
         _description = std::string("parameter subset for state in PF");
         _validity = "Names in parameter block separated by ,";
         _default = {};
         _default_value = {};
         _constrained = false;
         _value = _default_value;
      }
      bool is_valid(std::string value){
         return !std::regex_match(value, std::regex( (
    "((\\+|-)?[[:digit:]]+)(\\.(([[:digit:]]+)?))?" ) ) );
      }
   };
}
#endif
```

In order to register `state` and `old_state`, `arg_pf_old_state()` and `arg_pf_state()` must be added to `_subarguments` in the constructor of `arg_pf()`.

C.2.3 Initial Distribution

The initial distribution $q_0()$ should be described by another .stan file to be compiled by `stanc`, as if it was a standalone .stan model. The generated .hpp file should then be compiled together with the model .hpp file describing the target, along with the rest of Stan's back end by using `mpic++`. However this is not currently possible on CmdStan because all .hpp files generated by `stanc` end with a typedef instruction which transforms the arbitrary model name into an alias called `stan_model`, such that the rest of Stan's back end can access the model class in a general purpose way. Therefore, having two or more .hpp files generated by `stanc` to be compiled into the same executable file would result in a compilation error.

A solution to overcome this limitation would be to slightly change `stanc` to make it accept an extra optional compilation flag, named for example `--prior`. If this flag is given, the final typedef in the model .hpp file, `stanc` substitutes the model name with the alias `prior`, instead of `stan_model`. Then, the task `Initialise()` in Algorithm 4 is instructed to use `prior` to create a model object to generate the particles $\mathbf{x}_0$.

To achieve that, one first needs to add an extra boolean variable in `stanc_helper` which is found in CmdStan/stan/src/stan/command. This boolean variable has to be set to `true` or `false`, depending on whether the user has used the flag `prior` in the `stanc` compilation line. This can be done as follows:

```
bool prior = cmd.has_flag("prior");
```

After that, the boolean flag has to be passed on `compile` (called by `stanc_helper` and defined in compiler.hpp in the folder CmdStan/stan/src/stan/lang/), then passed on `generate_cpp` (called by `compile` and defined in generate_cpp.hpp in the folder CmdStan/stan/src/stan/lang/generator/) and finally passed on `generate_model_typedef` called by `generate_cpp`. The C++ code of `generate_model_typedef` should be changed as follows:

```
#ifndef STAN_LANG_GENERATOR_GENERATE_MODEL_TYPEDEF_HPP
#define STAN_LANG_GENERATOR_GENERATE_MODEL_TYPEDEF_HPP

#include <stan/lang/ast.hpp>
#include <stan/lang/generator/constants.hpp>
#include <ostream>
#include <string>

namespace stan {
   namespace lang {
      /**
      * Generate reusable typedef of <code>stan_model</code> for
      * specified model name writing to the specified stream.
      *
      * @param model_name name of model
      * @param o stream for generating
      */
      void generate_model_typedef(const std::string& model_name,
         std::ostream& o, const bool prior = false) {
         if(prior)
            o << "typedef " << model_name << "_namespace::" <<
               model_name << " prior;" << EOL2;
         else
            o << "typedef " << model_name << "_namespace::" <<
               model_name << " stan_model;" << EOL2;
```

```
        }
    }
}
#endif
```

At this point, some trivial changes have to be made to the makefile, but in this case details are omitted for brevity.

It is now actually possible to describe a dynamic model using regular Stan's syntax.

C.2.4 Proposal Distributions

Streaming-Stan also provides, for the PFs and Fixed-Lag SMC methods, multiple proposal distributions, as well as SMC-Stan does in the case of SMC sampler. These proposals are named FL-SMC, FL-HMC and FL-NUTS and are described in Section 6.3.1. The user decides which one to call by assigning a valid value to the argument `proposal`. Depending on this value, the IS step in Algorithm 4 is given a numerical flag, called `prop`, which is checked within a switch-break statement such that the correct sampling and weighting equations can be used. The value of `prop` is modulated by a series of nested if-else statements within the body of instructions in command.hpp which is executed if `algo->value()` is equal to `pf`.

The implementation strategy is the same as in SMC-Stan. More precisely, for each proposal distribution, a new class has to be created and added to `_value` in the constructor of `arg_proposal`, in order to register a new valid keyword for the argument `proposal`. These classes and keywords are named: `arg_fl_smc_proposal` and `FL-SMC` for the FL-SMC proposal, `arg_fl_hmc_proposal` and `FL-HMC` for the FL-HMC proposal, and `arg_fl_nuts_proposal` and `FL-NUTS` for the FL-NUTS proposal.

C.2.4.1 Fixed-Lag SMC

Equations (2.26) a (2.28) describe FL-SMC. These equations are function of the lag l which should be given as input. Therefore, it is necessary to create a new class called `arg_lag`, child to `singleton_argument` in the form of integer argument, having keyword `Lag` and default value 0. The C++ code for `arg_lag` is omitted for brevity, but it is equivalent to the one of any other integer arguments, such as `arg_T`. Then, `arg_lag` has to be added to `_subarguments` of the constructor of `arg_fl_smc_proposal`. Hence, the C++ for this class is omitted for brevity, but the reader is referred to Section C.1.1.2 to see how the proposal classes are defined in SMC-Stan.

C.2.4.2 Fixed-Lag HMC

To use this proposal, the following `arg_fl_hmc_proposal` class has to be added to CmdStan/src/cmdstan/arguments and `arg_fl_hmc_proposal()` has to be added to `_values` in the constructor of `arg_proposal`. The C++ code for this class is omitted for brevity but is equivalent to the one for `arg_hmc_proposal` in SMC-Stan (see Section

C.1.1.2), apart from having _name equal to the keyword FL-HMC and one extra input in _subarguments for arg_lag.

C.2.4.3 Fixed-Lag NUTS

The class arg_fl_nuts_proposal is equivalent to arg_fl_hmc_proposal apart from having FL-NUTS assigned to _name and not having arg_num_leapfrog_steps() added to _subarguments, as the number of leapfrog steps in NUTS is adaptive. To use FL-NUTS, arg_fl_nuts_proposal() must also be added to _value in the constructor of arg_proposal.

C.2.5 Examples

This section shows how to use Streaming-Stan to describe and solve a wide range of dynamic models.

C.2.5.1 Stochastic Volatility

The first example is the Stochastic Volatility model which is first illustrated in this thesis in Section 3.4.5.2. This is a relatively easy model to describe in Streaming-Stan, as the state-transition equation is single-dimensional and linear. The following is suitable .stan file to implement this model:

```
data {
   real phi; real beta; real sigma;
   real Xt_1;
   real Yt;
}
parameters {
   real Vt;
}
transformed parameters {
   real Xt;
   Xt = phi*Xt_1 + sigma*Vt
}
model {
   target += normal_lpdf(Yt | beta*exp(0.5*Xt) 1); //likelihood
   target += normal_lpdf(Xt | phi*Xt_1, pow(sigma, 2)); //prior
}
```

and the initial distribution may be described by the following .stan file:

```
data {
   real phi; real sigma;
}
parameters {
   real V0;
}
transformed parameters {
   real X0;
   X0 = sqrt(pow(sigma, 2)/(1 - pow(phi, 2)))*V0;
}
model {
   target += normal_lpdf(X0 | 0, pow(sigma, 2)/(1 - pow(phi, 2)));
}
```

As we can see, this model samples process noise from $\mathcal{N}(0, 1)$ which is then rescaled in the **transformed parameters** block where the state is defined and computed.

The data has to be written in a separate file. Streaming-Stan, as well as Stan and SMC-Stan, accepts different extensions, such as .R files or .json files. The following is a suitable .R data file for this model:

```
phi <- 0.9731
beta <- 0.6338
sigma <- 0.1726
Xt_1 <- 0.0
Yt <- 0.0
```

where the $\mathbf{X}_{t-1}$ and $\mathbf{Y}_t$ can be initialised to any arbitrary value, since $\mathbf{X}_{t-1}$ is overwritten at run-time by $\mathbf{X}_t$ and $\mathbf{Y}_t$ is expected to be updated at run-time by an external measurement source, e.g. by an external software that edits the content of the data file.

If the executable and data files are named, for example, sv and sv.data.R, then it is possible to execute the model by running the following command:

```
$ mpirun -np 1 sv method=sample algorithm=pf proposal=FL-NUTS \
stepsize=0.01 old_state=Xt_1 state=Xt Lag=0 T=1 Tpf=100 \
num_samples=512 data file=sv.data.R
```

which generates 100 estimates by sampling $N = 512$ particles at each time step using FL-NUTS as proposal.

C.2.5.2 Flood Water Level

The second example is the flood water level model presented in Section 6.4.1.2. The following, called water.stan, is a suitable .stan file to describe this model in Streaming-Stan.

```
data {
   int t;
   real Xt_1;
   real Yt;
}
parameters {
   real Vt;
}
```

```
transformed parameters {
   real Xt;
   Xt = 0.5*Xt_1 + 25*Xt_1/(1+pow(Xt_1, 2)) + cos(1.2*t) + Vt
}
model {
   target += normal_lpdf(Yt | 0.05*pow(Xt, 2), 1); //likelihood
   target += normal_lpdf(Xt | 0.5*Xt_1 + 25*Xt_1/(1+pow(Xt_1, 2)) +
      cos(1.2*t), 1); //prior
}
```

where the time step t is used as data since it is expected to be incremented by the data/measurement source. The code for the initial distribution follows:

```
parameters {
   real X0;
}
transformed parameters {
   real X0;
   X0 = V0;
}
model {
   target += normal_lpdf(X0 | 0, 1);
}
```

The .R data file, called water.data.R, is

```
t <- 0
Xt_1 <- 0.0
Yt <- 0.0
```

The same command line as in the previous section can be used to run this model, apart from changes on the executable and data file names. The data source has been coded to update water.data.R every second with incremental values of t, and new values of $\mathbf{Y}_t$ which can downloaded from https://www.water.gov.my, as done in [73, 74].

C.2.5.3 Bearing-Only Tracking

The next example is the bearing-only tracking model presented in Section 6.4.1.3.

```
data {
   matrix[4, 4] A; //State transition matrix
   matrix[4, 4] sigma; //Process noise sigma up to a constant factor
   real q; //constant term in process noise covariance
   int My; //number of measurements
   real p[My*2]; //sensors' positions
   real W; //measurement noise standard deviation
   vector[4] Xt_1; //Old state
   real Yt[My]; //measurements
}
transformed data {
   matrix[4, 4] Sigma; //Full process noise covariance
   Sigma = q*sigma;
}
parameters {
   vector[4] Vt; //Process noise
}
transformed parameters{ //State transition
   vector[4] Xt;
   Xt = A*Xt_1 + sqrt(Sigma)*Vt;
}
model { //Target p(Y_t | X_t) * p(X_t | X_{t-1})
   real pred_y[My]; //predicted measurement
   for(k in 1:My){
      pred_y[k] = atan((Xt[1] - p[k*2 - 1]) / (Xt[3] - p[k*2]));
   }
   target += normal_lpdf(Yt | pred_y, W); //likelihood
   target += multi_normal_lpdf(Xt | A*Xt_1, Sigma); //prior
}
```

where the full process noise covariance is computed in the **transformed data** block and can be modulated at run-time by the scalar q.

The code for the initial distribution follows:

```
parameters {
   real V0;
}
transformed parameters {
   real X0;
   X0 = V0;
}
model {
   target += normal_lpdf(X0 | 0, 1);
}
```

The .R data file, called BO_tracking.data.R, is the following:

```
A <- structure(c(1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0,
    1.0, 0.0, 0.0, 0.0, 1.0), .Dim = c(4, 4))
Sigma <- structure(c(0.3333, 0.5, 0, 0, 0.5, 1, 0, 0, 0, 0, 0.3333,
    0.5, 0, 0, 0.5, 1), .Dim = c(4, 4))
q <- 5.0
num_sensors <- 4
p <- c(0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 1.0)
W <- 0.333
Xt_1 <- c(1.0, 0.2, 1.0, 0.2)
```

```
Yt <- c(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
```

The command line to run is again the same as in the previous sections, apart from using the names of the execution and data files.

C.2.5.4 Vacuum Arc Remelting

The final example of this section is the VAR model described in Section 4.4.5.3. A suitable model .stan file, called var.stan, could be:

```
data {
   real Cdd; real ar; real Cdp; real Ae; real hm; real G11;
   real Csd; real a; real G21; real rho; real Csp; real G41;
   real dt; real Ac; real Bdc; real Bic; real Buc; real Bhec;
   real Am; real Bdm; real Bim; real Bum; real Bhem; real Vc;
   real Ri; real Vram; real Ic; real dtau;
   vector[9] Q;
   vector[8] R;
   vector[9] Sigma;
   vector[9] Xnom;
   vector[9] Xt_1;
   vector[8] Yt;
}
parameters {
   real dI;
   real dVram;
   real du;
   real dhe;
}
transformed parameters {
   vector[9] Xt;
   Xt[1] = Xt_1[1] + dt*(Cdd*ar/Xt_1[1] +
   Xt_1[5]*(Vc+Ri*Xt_1[9])*Cdp/(Ae*hm)) + G11*20*dt*dI;
   Xt[2] = Xt_1[2] + dt*(-Csd*a*ar/Xt_1[1] +
   Xt_1[5]*(Vc+Ri*Xt_1[9])*Cdp*a/(Ae*hm) - Vram) +
      G21*20*dt*dI - dt*dVram;
   Xt[3] = Xt_1[3] + dt*(Csd*Ae*rho/Xt_1[1] +
   Xt_1[5]*(Vc+Ri*Xt_1[9])*Csp*rho/hm) + G41*20*dt*dI;
   Xt[4] = Xt_1[4] + Vram*dt + dt*dVram;
   Xt[5] = Xt_1[5] + 0.001*pow(Xnom[5],2)*sqrt(dt)*du;
   Xt[6] = Xt_1[6] - dt*(Ac*(Xt_1[6] - Xnom[6]) - Bdc*(Xt_1[1]-Xnom[1])
      - Bic*(Xt_1[9]-Xnom[9]) - Buc*(Xt_1[5]-Xnom[5]) -
      Bhec*(Xt_1[8]-Xnom[8])) + Bic*20*dt*dI;
   Xt[7] = Xt_1[7] - dt*(Am*(Xt_1[7] - Xnom[7]) - Bdm*(Xt_1[1]-Xnom[1])
      - Bim*(Xt_1[9]-Xnom[9]) - Bum*(Xt_1[5]-Xnom[5]) -
      Bhem*(Xt_1[8]-Xnom[8])) + Bim*20*dt*dI;
   Xt[8] = Xt_1[8] + 0.001*pow(Xnom[8],2)*sqrt(dt)*dhe;
   Xt[9] = Ic + (Ic - Xnom[9])*exp(-dt/dtau) + 20*dt*dI;
}
model {
   vector[8] pred_y;
   for(i in 1:7){
      pred_y[i] = Xt[i+1];
   }
   pred_y[8] = Vc + Ri*Xt[9];
   vector[9] pred_X;
   pred_X[1] = Xt_1[1] + dt*(Cdd*ar/Xt_1[1] +
      Xt_1[5]*(Vc+Ri*Xt_1[9])*Cdp/(Ae*hm));
   pred_X[2] = Xt_1[2] + dt*(-Csd*a*ar/Xt_1[1] +
      Xt_1[5]*(Vc+Ri*Xt_1[9])*Cdp*a/(Ae*hm) - Vram);
   pred_X[3] = Xt_1[3] + dt*(Csd*Ae*rho/Xt_1[1] +
      Xt_1[5]*(Vc+Ri*Xt_1[9])*Csp*rho/hm);
   pred_X[4] = Xt_1[4] + Vram*dt;
   pred_X[5] = Xt_1[5];
   pred_X[6] = Xt_1[6] - dt*(Ac*(Xt_1[6] - Xnom[6]) -
    Bdc*(Xt_1[1]-Xnom[1])
      - Bic*(Xt_1[9]-Xnom[9]) - Buc*(Xt_1[5]-Xnom[5]) -
      Bhec*(Xt_1[8]-Xnom[8]));
   pred_X[7] = Xt_1[7] - dt*(Am*(Xt_1[7] - Xnom[7]) -
    Bdm*(Xt_1[1]-Xnom[1])
      - Bim*(Xt_1[9]-Xnom[9]) - Bum*(Xt_1[5]-Xnom[5]) -
      Bhem*(Xt_1[8]-Xnom[8]));
   pred_X[8] = Xt_1[8];
   pred_X[9] = Ic + (Ic - Xnom[9])*exp(-dt/dtau);
   target += multi_normal_lpdf(Yt | pred_y, diag_matrix(R));
   target += multi_normal_lpdf(Xt | pred_X, diag_matrix(Sigma));
}
```

and the file for the initial distribution follows:

```
data {
   vector[9] Q;
   vector[9] Xnom;
}
parameters {
   vector[9] v0;
}
transformed parameters {
   vector[9] X0;
   for(i in 1:9){
      X0[i] = Xnom[i] + sqrt(Q[i])*v0[i];
   }
}
model {
    target += multi_normal_lpdf(X0 | Xnom, diag_matrix(Q));
}
```

The data file is omitted for brevity due to its large size, but the constant values are found in Tables 4.2, 4.3 and 4.4 and the value of `Vram` and `Yt` are changed at run-time by an external measurement source. The compilation and running commands are the same as in the previous example, apart from changes to the file names and input arguments.

Appendix D

Reversible and Symplectic Numerical Integrators: Properties

This appendix proves a fundamental property of Leapfrog related to the Jacobian matrix of the forward and backward integration. This property is crucial to design FL-HMC and FL-NUTS, the two novel proposal distributions for FL-SMC methods presented in Chapter 6. The equations describing Leapfrog are found in (2.32), but for this proof it is more convenient to use the following alternative version of the same:

$$\mathbf{X}_t = \mathbf{X}_{t-1} + \Delta h \mathbf{V}_{t-1} + \frac{1}{2}\Delta h^2 \mathbf{A}_t \tag{D.1a}$$

$$\mathbf{V}_t = \mathbf{V}_{t-1} + \frac{1}{2}\Delta h(\mathbf{A}_{t-1} + \mathbf{A}_t) \tag{D.1b}$$

where $\mathbf{X}_{t-1}$ and $\mathbf{V}_{t-1}$ are the starting point and momentum, $\mathbf{X}_t$ and $\mathbf{V}_t$ are the final point and momentum, Δh is the integration step, and $\mathbf{A}_{t-1}$ and $\mathbf{A}_t$ are the initial and final accelerations terms. The property to prove is expressed by the following theorem.

Theorem D.1. *Let* $\mathbf{f}$ *represent the Leapfrog integrator, such that* $\mathbf{f}(\mathbf{X}_{t-1}, \mathbf{V}_{t-1}) = (\mathbf{X}_t, \mathbf{V}_t)$*, and* $\mathbf{J}(\mathbf{X}_{t-1}, \mathbf{V}_{t-1}) = \mathbf{J}^{for}$ *be the Jacobian matrix associated to this transition. Leapfrog is time reversible, which means that if* $\mathbf{f}(\mathbf{X}_{t-1}, \mathbf{V}_{t-1}) = (\mathbf{X}_t, \mathbf{V}_t)$*, by starting from* $\mathbf{X}_t$ *with reversed final momentum* $-\mathbf{V}_t$ *we get back to* $\mathbf{X}_{t-1}$*, i.e.* $\mathbf{f}(\mathbf{X}_t, -\mathbf{V}_t) = (\mathbf{X}_{t-1}, \mathbf{V}_{t-1})$*. Hence, let* $\mathbf{J}(\mathbf{X}_t, -\mathbf{V}_t) = \mathbf{J}^{back}$ *be the Jacobian matrix related to the backward integration. For Leapfrog, it is possible to show that the determinants of the forward and backward Jacobian matrices have the same absolute value:*

$$||\mathbf{J}(\mathbf{X}_{t-1}, \mathbf{V}_{t-1})|| = ||\mathbf{J}(\mathbf{X}_t, -\mathbf{V}_t)|| \tag{D.2}$$

Proof of Theorem D.1. To prove (D.2), we need to start from the expression for the forward Jacobian matrix:

$$\mathbf{J}^{for} = \begin{bmatrix} \frac{\partial \mathbf{X}_t}{\partial \mathbf{X}_{t-1}} & \frac{\partial \mathbf{X}_t}{\partial \mathbf{V}_{t-1}} \\ \frac{\partial \mathbf{V}_t}{\partial \mathbf{X}_{t-1}} & \frac{\partial \mathbf{V}_t}{\partial \mathbf{V}_{t-1}} \end{bmatrix} = \begin{bmatrix} \mathbf{I} + \frac{\Delta h^2}{2}\frac{\partial \mathbf{A}_t}{\partial \mathbf{X}_{t-1}} & \Delta h \mathbf{I} \\ \frac{\Delta h}{2}\left(\frac{\partial \mathbf{A}_t}{\partial \mathbf{X}_{t-1}} + \frac{\partial \mathbf{A}_t}{\partial \mathbf{X}_t} + \frac{\Delta h^2}{2}\frac{\partial \mathbf{A}_t}{\partial \mathbf{X}_t}\frac{\partial \mathbf{A}_t}{\partial \mathbf{X}_{t-1}}\right) & \mathbf{I} + \frac{\Delta h^2}{2}\frac{\partial \mathbf{A}_t}{\partial \mathbf{X}_t} \end{bmatrix} \tag{D.3}$$

The determinant of this matrix has been shown to be equal to 1 [98], meaning that Leapfrog is also a symplectic numerical integrator. For Leapfrog, which is time reversible, it is also possible to show that the backward Jacobian matrix can be computed by transposing each block of the matrix $\mathbf{J}^{for}$:

$$\mathbf{J}^{back} = \begin{bmatrix} \left(\frac{\partial \mathbf{X}_t}{\partial \mathbf{X}_{t-1}}\right)^{\mathsf{T}} & \left(\frac{\partial \mathbf{X}_t}{\partial \mathbf{V}_{t-1}}\right)^{\mathsf{T}} \\ \left(\frac{\partial \mathbf{V}_t}{\partial \mathbf{X}_{t-1}}\right)^{\mathsf{T}} & \left(\frac{\partial \mathbf{V}_t}{\partial \mathbf{V}_{t-1}}\right)^{\mathsf{T}} \end{bmatrix} \tag{D.4}$$

In other words, if $\mathbf{A}$ is an arbitrary block matrix and we let α be the operation:

$$\alpha(\mathbf{A}) = \alpha\left(\begin{bmatrix} \mathbf{A}^{1,1} & \mathbf{A}^{1,2} \\ \mathbf{A}^{2,1} & \mathbf{A}^{2,2} \end{bmatrix}\right) = \begin{bmatrix} \left(\mathbf{A}^{1,1}\right)^{\mathsf{T}} & \left(\mathbf{A}^{1,2}\right)^{\mathsf{T}} \\ \left(\mathbf{A}^{2,1}\right)^{\mathsf{T}} & \left(\mathbf{A}^{2,2}\right)^{\mathsf{T}} \end{bmatrix} \tag{D.5}$$

we can then say that:

$$\mathbf{J}^{back} = \alpha(\mathbf{J}^{for}) \tag{D.6}$$

Now, we need to consider the generic scenario, where the integration $\mathbf{f}(\mathbf{X}_{t-1}, \mathbf{V}_{t-1}) = (\mathbf{X}_t, \mathbf{V}_t)$ is the result of L Leapfrog steps, such that we have a sequence of position-momentum pairs $(\mathbf{X}_{t-1}, \mathbf{V}_{t-1}), (\mathbf{X}^1, \mathbf{V}^1), (\mathbf{X}^2, \mathbf{V}^2), ..., (\mathbf{X}^L, \mathbf{V}^L) = (\mathbf{X}_t, \mathbf{V}_t)$. In this case, the forward Jacobian for the full forward integration is given by the following matrix product:

$$\mathbf{J}^{for} = \mathbf{J}^{for}_{L-1} \cdot \mathbf{J}^{for}_{L-2} \cdot \mathbf{J}^{for}_{L-3} \cdots \mathbf{J}^{for}_{1} \cdot \mathbf{J}^{for}_{0} \tag{D.7}$$

while the backward Jacobian for the full backward integration is computed as in the following matrix product:

$$\mathbf{J}^{back} = \mathbf{J}^{back}_{0} \cdot \mathbf{J}^{back}_{1} \cdot \mathbf{J}^{back}_{2} \cdots \mathbf{J}^{back}_{L-2} \cdot \mathbf{J}^{back}_{L-1} \tag{D.8}$$

where by (D.6) $\mathbf{J}^{back}_{i+1} = \alpha(\mathbf{J}^{for}_{i})$.

It is relatively straightforward to prove that for a block matrix $\mathbf{A} = \mathbf{B}^0 \cdot \mathbf{B}^1 \cdot \mathbf{B}^2 \cdots \mathbf{B}^{L-2} \cdot \mathbf{B}^{L-1}$, we have:

$$\begin{aligned} \alpha(\mathbf{A}) &= \alpha(\mathbf{B}^0 \cdot \mathbf{B}^1 \cdot \mathbf{B}^2 \cdots \mathbf{B}^{L-2} \cdot \mathbf{B}^{L-1}) \\ &= \alpha(\mathbf{B}^{L-1}) \cdot \alpha(\mathbf{B}^{L-2}) \cdot \alpha(\mathbf{B}^{L-3}) \cdots \alpha(\mathbf{B}^{1}) \cdot \alpha(\mathbf{B}^{0}) \end{aligned} \tag{D.9}$$

By applying (D.9) to (D.7), we obtain:

$$\begin{aligned} \alpha(\mathbf{J}^{for}) &= \alpha(\mathbf{J}^{for}_0) \cdot \alpha(\mathbf{J}^{for}_1) \cdot \alpha(\mathbf{J}^{for}_2) \cdots \alpha(\mathbf{J}^{for}_{L-2}) \cdot \alpha(\mathbf{J}^{for}_{L-1}) \\ &= \mathbf{J}^{back}_0 \cdot \mathbf{J}^{back}_1 \cdot \mathbf{J}^{back}_2 \cdots \mathbf{J}^{back}_{L-2} \cdot \mathbf{J}^{back}_{L-1} \\ &= \mathbf{J}^{back} \end{aligned} \tag{D.10}$$

which means (D.6) can be applied to the full integrator. In particular, (D.10) automatically proves (D.2). $\square$

Bibliography

[1] K. Achutegui and J. Míguez, *A Parallel Resampling Scheme and Its Application to Distributed Particle Filtering in Wireless Networks*, 2011 4th IEEE International Workshop on Computational Advances in Multi-Sensor Adaptive Processing (CAMSAP), 2011, pp. 81–84.

[2] M. Ajtai, J. Komlós, and E. Szemerédi, *An 0(N log N) Sorting Network*, Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing (New York, NY, USA), STOC '83, ACM, 1983, pp. 1–9.

[3] P. Akshay, D. Vrushabh, K. Sonam, S. Wagh, and N. Singh, *Hamiltonian Monte Carlo Based Path Integral for Stochastic Optimal Control*, 2020 28th Mediterranean Conference on Control and Automation (MED), 2020, pp. 254–259.

[4] M. Al-Hashimi, M. Saleh, O. Abulnaja, and N. Aljabri, *On the Power Characteristics of Mergesort: An Empirical Study*, 2017 Intl Conf on Advanced Control Circuits Systems (ACCS) Systems 2017 Intl Conf on New Paradigms in Electronics Information Technology (PEIT), 2017, pp. 172–178.

[5] Anthony Vinay Kumar S and A. Arya, *Fastbit-Radix Sort: Optimized Version of Radix Sort*, 2016 11th International Conference on Computer Engineering Systems (ICCES), 2016, pp. 305–312.

[6] 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.

[7] S.W.A.H. Baddar and K.E. Batcher, *Designing Sorting Networks: A New Paradigm*, SpringerLink : Bücher, Springer New York, 2012.

[8] F. Bai, F. Gu, X. Hu, and S. Guo, *Particle Routing in Distributed Particle Filters for Large-Scale Spatial Temporal Systems*, IEEE Transactions on Parallel and Distributed Systems **27** (2016), no. 2, 481–493.

[9] Kenneth E. Batcher, *Sorting Networks and Their Applications*, Proceedings of the 1968 Spring Joint Computern Conference (SJCC), vol. 32, AFIPS Press, 1968, pp. 307–314.

[10] O. Ö. Bilgin and M. Demirekler, *Multi Mode Projectile Tracking With Marginalized Particle Filter*, 2015 IEEE Radar Conference, 2015, pp. 224–229.

[11] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman, *Pyro: Deep Universal Probabilistic Programming*, J. Mach. Learn. Res. **20** (2019), no. 1, 973–978.

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

[13] Miodrag Bolic, Akshay Athalye, Sangjin Hong, and Petar Djuric, *Study of Algorithmic and Architectural Characteristics of Gaussian Particle Filters*, Signal Processing Systems **61** (2010), 205–218.

[14] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux, *API Design for Machine Learning Software: Experiences From the Scikit-learn Project*, ECML PKDD Workshop: Languages for Data Mining and Machine Learning, 2013, pp. 108–122.

[15] J. Bull, James P. Enright, X. Guo, C. Maynard, and F. Reid, *Performance Evaluation of Mixed-Mode OpenMP/MPI Implementations*, International Journal of Parallel Programming **38** (2010), 396–417.

[16] Mark Bull, *MPI and OpenMP*, Lecture slides.

[17] L. Chaari, J. Tourneret, and H. Batatia, *A General Non-Smooth Hamiltonian Monte Carlo Scheme Using Bayesian Proximity Operator Calculation*, 2017 25th European Signal Processing Conference (EUSIPCO), 2017, pp. 1220–1224.

[18] L. Chaari, J. Tourneret, C. Chaux, and H. Batatia, *A Hamiltonian Monte Carlo Method for Non-Smooth Energy Sampling*, IEEE Transactions on Signal Processing **64** (2016), no. 21, 5585–5594.

[19] Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha, *Scan Primitives for Vector Computers*, Proceedings of the 1990 ACM/IEEE Conference on Supercomputing, Supercomputing '90, IEEE Computer Society Press, 1990, pp. 666–675.

[20] N. Chopin, P. E. Jacob, and O. Papaspiliopoulos, *SMC^2: An Efficient Algorithm for Sequential Analysis of State Space Models*, Journal of the Royal Statistical Society: Series B (Statistical Methodology) **75** (2013), no. 3, 397–426.

[21] P. Closas and M. F. Bugallo, *Improving Accuracy by Iterated Multiple Particle Filtering*, IEEE Signal Processing Letters **19** (2012), no. 8, 531–534.

[22] José Mir Costa, *Estimation of Tumor Size Evolution Using Particle Filters*, Journal of Computational Biology (2015).

[23] R. Daviet, *Inference With Hamiltonian Sequential Monte Carlo Simulators*, Risk Management & Analysis in Financial Institutions eJournal (2016).

[24] R. Delgado-Gonzalo, N. Chenouard, and M. Unser, *A New Hybrid Bayesian-Variational Particle Filter With Application to Mitotic Cell Tracking*, 2011 IEEE International Symposium on Biomedical Imaging: From Nano to Macro, March 2011, pp. 1917–1920.

[25] Omer Demirel, Ihor Smal, W.J. Niessen, Erik Meijering, and Ivo Sbalzarini, *PPF - A Parallel Particle Filtering Library*, IET Conference Publications **2014** (2013).

[26] Petar Djuric, Ting Lu, and Monica Bugallo, *Multiple Particle Filtering*, vol. 3, 05 2007, pp. III–1181.

[27] P. M. Djurić and M. F. Bugallo, *Multiple Particle Filtering With Improved Efficiency and Performance*, 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2015, pp. 4110–4114.

[28] A. Doucet and S. Sénécal, *Fixed-Lag Sequential Monte Carlo*, 2004 12th European Signal Processing Conference, 2004, pp. 861–864.

[29] Arnaud Doucet, Mark Briers, and Stéphane Sénécal, *Efficient Block Sampling Strategies for Sequential Monte Carlo Methods*, Journal of Computational and Graphical Statistics **15** (2006), no. 3, 693–711.

[30] Arnaud Doucet and Adam Johansen, *A Tutorial on Particle Filtering and Smoothing: Fifteen Years Later*, Handbook of Nonlinear Filtering **12** (2009).

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

[32] Jeff Edmonds, *How to Think About Algorithms*, 01 2008.

[33] S. Geman and D. Geman, *Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images*, IEEE Transactions on Pattern Analysis and Machine Intelligence **PAMI-6** (1984), no. 6, 721–741.

[34] G. Graditi, R. Ciavarella, M. Valenti, A. Bracale, and P. Caramia, *Advanced Forecasting Method to the Optimal Management of a DC Microgrid in Presence of Uncertain Generation*, 2015 International Conference on Renewable Energy Research and Applications (ICRERA), 2015, pp. 1586–1590.

[35] Fredrik Gustafsson, *Particle Filter Theory and Practice With Positioning Applications*, IEEE Aerospace and Electronic Systems Magazine **25** (2010), no. 7, 53–82.

[36] Fabiana C. Hamilton, Marcelo J. Colaço, Rogério N. Carvalho, and Albino J.K. Leiroz, *Heat Transfer Coefficient Estimation of an Internal Combustion Engine Using Particle Filters*, Inverse Problems in Science and Engineering **22** (2014), no. 3, 483–506.

[37] W. K. Hastings, *Monte Carlo Sampling Methods Using Markov Chains and Their Applications*, Biometrika **57** (1970), no. 1, 97–109.

[38] A. Hegyi, L. Mihaylova, R. Boel, and Z. Lendek, *Parallelized Particle Filtering for Freeway Traffic State Tracking*, 2007 European Control Conference (ECC), 2007, pp. 2442–2449.

[39] Kari Heine, Nick Whiteley, and A.Taylan Cemgil, *Parallelizing Particle Filters With Butterfly Interactions*, Scandinavian Journal of Statistics **47** (2020), no. 2, 361–396.

[40] Gustaf Hendeby, Rickard Karlsson, and Gustafsson Fredrik, *Particle Filtering: The Need for Speed*, EURASIP Journal on Advances in Signal Processing **2010** (2010).

[41] M. D. Hill and M. R. Marty, *Amdahl's Law in the Multicore Era*, Computer **41** (2008), no. 7, 33–38.

[42] C. A. R. Hoare, *Quicksort*, The Computer Journal **5** (1962), no. 1, 10–16.

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

[44] Matthew D. Homan and Andrew Gelman, *The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo*, J. Mach. Learn. Res. **15** (2014), no. 1, 1593–1623.

[45] Richard E. Ladner and Michael J. Fischer, *Parallel Prefix Computation*, J. ACM **27** (1980), no. 4, 831–838.

[46] David Last, Paul Thomas, Steven Hiscocks, Jordi Barr, David Kirkland, Mamoon Rashid, Sang Bin Li, and Lyudmil Vladimirov, *Stone Soup: Announcement of Beta Release of an Open-Source Framework for Tracking and State Estimation*, Signal Processing, Sensor/Information Fusion, and Target Recognition XXVIII (Ivan Kadar, Erik P. Blasch, and Lynne L. Grewe, eds.), vol. 11018, International Society for Optics and Photonics, SPIE, 2019, pp. 52 – 63.

[47] Anthony Lee, Christopher Yau, Michael B. Giles, Arnaud Doucet, and Christopher C. Holmes, *On the Utility of Graphics Cards to Perform Massively Parallel Simulation of Advanced Monte Carlo Methods*, Journal of Computational and Graphical Statistics **19** (2010), no. 4, 769–789.

[48] 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.

[49] Tiancheng Li, G. Villarrubia, Shu-dong Sun, Juan Corchado Rodríguez, and Javier Bajo, *Resampling Methods for Particle Filtering: Identical Distribution, a New Method, and Comparable Study*, Frontiers of Information Technology & Electronic Engineering **16** (2015), 969–984.

[50] Richard J. Lipton and Kenneth W. Regan, *People, Problems, and Proofs: Essays From Gdel's Lost Letter 2010*, Springer Publishing Company, Incorporated, 2013.

[51] Hedibert Lopes and Ruey Tsay, *Particle Filters and Bayesian Inference in Financial Econometrics*, Journal of Forecasting **30** (2011), 168–209.

[52] F. Lopez, L. Zhang, J. Beaman, and A. Mok, *Implementation of a Particle Filter on a GPU for Nonlinear Estimation in a Manufacturing Remelting Process*, 2014 IEEE/ASME International Conference on Advanced Intelligent Mechatronics, July 2014, pp. 340–345.

[53] Felipe Lopez, Lixun Zhang, Aloysius Mok, and Joseph Beaman, *Particle Filtering on GPU Architectures for Manufacturing Applications*, Computers in Industry **71** (2015), 116 – 127.

[54] J.H. Lowenstein, *Essentials of Hamiltonian Dynamics*, Essentials of Hamiltonian Dynamics, Cambridge University Press, 2012.

[55] P. Mann, *Lagrangian and Hamiltonian Dynamics*, Oxford University Press, 2018.

[56] Luca Martino, Jesse Read, Víctor Elvira, and Francisco Louzada, *Cooperative Parallel Particle Filters for Online Model Selection and Applications to Urban Mobility*, Digital Signal Processing **60** (2017), 172 – 185.

[57] 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, May 2012, pp. 1–8.

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

[59] L. Mihaylova, A. Hegyi, A. Gning, and R. K. Boel, *Parallelized Particle and Gaussian Sum Particle Filters for Large-Scale Freeway Traffic Systems*, IEEE Transactions on Intelligent Transportation Systems **13** (2012), no. 1, 36–48.

[60] Y. Mitani, F. Ino, and K. Hagihara, *Parallelizing Exact and Approximate String Matching via Inclusive Scan on a GPU*, IEEE Transactions on Parallel and Distributed Systems **28** (2017), no. 7, 1989–2002.

[61] Pierre Del Moral, Arnaud Doucet, and Ajay Jasra, *Sequential Monte Carlo Samplers*, Journal of the Royal Statistical Society. Series B (Statistical Methodology) **68** (2006), no. 3, 411–436.

[62] Lawrence M. Murray, Anthony Lee, and Pierre E. Jacob, *Parallel Resampling in the Particle Filter*, Journal of Computational and Graphical Statistics **25** (2016), no. 3, 789–805.

[63] C. A. Naesseth, F. Lindsten, and T. B. Schön, *High-Dimensional Filtering Using Nested Sequential Monte Carlo*, IEEE Transactions on Signal Processing **67** (2019), no. 16, 4177–4188.

[64] Radford M. Neal, *Slice Sampling*, The Annals of Statistics **31** (2003), no. 3, 705–741.

[65] 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.

[66] Frank Nielsen, *Introduction to HPC with MPI for Data Science*, 09 2016.

[67] M. Nishio and A. Arakawa, *Performance of Hamiltonian Monte Carlo and No-U-Turn Sampler for Estimating Genetic Parameters and Breeding Values*, Genetics, Selection, Evolution : GSE **51** (2019).

[68] Metropolis NS, A.W. Rosenbluth, M.N. Rosenbluth, AH Teller, and E J. Teller, *Equation of State Calculations by Fast Computing Machines*, **21** (1953), 1087–1092.

[69] G. Özen, S. Atzeni, M. Wolfe, A. Southwell, and G. Klimowicz, *OpenMP GPU Offload in Flang and LLVM*, 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC), Nov 2018, pp. 1–9.

[70] M. S. Paterson, *Improved Sorting Networks With o(logN) Depth*, Algorithmica **5** (1990), no. 1, 75–92.

[71] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, *Scikit-learn: Machine Learning in Python*, Journal of Machine Learning Research **12** (2011), 2825–2830.

[72] P. Pileggi, M. Bocquel, and M. Podt, *Integrated Processing for Extended Target Tracking Using Sequential Hamiltonian Monte Carlo*, 2017 20th International Conference on Information Fusion (Fusion), 2017, pp. 1–8.

[73] F. A. Ruslan, R. Adnan, A. M. Samad, and Z. M. Zain, *Parameters Effect in Sampling Importance Resampling (SIR) Particle Filter Prediction and Tracking of Flood Water Level Performance*, 2012 12th International Conference on Control, Automation and Systems, 2012, pp. 868–872.

[74] F. A. Ruslan, Z. M. Zain, R. Adnan, and A. M. Samad, *Flood Water Level Prediction and Tracking Using Particle Filter Algorithm*, 2012 IEEE 8th International Colloquium on Signal Processing and its Applications, 2012, pp. 431–435.

[75] John Salvatier, Thomas V. Wiecki, and Christopher Fonnesbeck, *Probabilistic Programming in Python using PyMC3*, PeerJ Computer Science **2** (2016), e55.

[76] T. Schon, F. Gustafsson, and P. . Nordlund, *Marginalized Particle Filters for Mixed Linear/Nonlinear State-Space Models*, IEEE Transactions on Signal Processing **53** (2005), no. 7, 2279–2289.

[77] Joel Seiferas, *Sorting Networks of Logarithmic Depth, Further Simplified*, Algorithmica **53** (2009), no. 3, 374–384.

[78] F. Septier and G. W. Peters, *Langevin and Hamiltonian Based Sequential mcmc for Efficient Bayesian Filtering in High-Dimensional Spaces*, IEEE Journal of Selected Topics in Signal Processing **10** (2016), no. 2, 312–327.

[79] Hamid Shariati, Hassan Moosavi, and Mohammad Danesh, *Application of Particle Filter Combined With Extended Kalman Filter in Model Identification of an Autonomous Underwater Vehicle Based on Experimental Data*, Applied Ocean Research **82** (2019), 32 – 40.

[80] Steven S. Skiena, *Sorting and Searching*, pp. 103–144, Springer London, London, 2008.

[81] K. Stokfiszewski, D. Puchala, and M. Yatsymirskyy, *Effectiveness of GPU Realizations of Parallel Prefix-Sums Computation Algorithms*, 2018 IEEE 13th International Scientific and Technical Conference on Computer Sciences and Information Technologies (CSIT), vol. 1, 2018, pp. 436–439.

[82] Ingvar Strid, *Parallel Particle Filters for Likelihood Evaluation in DSGE Models: An Assessment*, Computing in Economics and Finance 2006 395, Society for Computational Economics, July 2006.

[83] 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.

[84] Stan Development Team, *Stan User's Guide Version 2.25*, (2019).

[85] Jeyarajan Thiyagalingam, Lykourgos Kekempanos, and Simon Maskell, *Mapreduce Particle Filtering With Exact Resampling and Deterministic Runtime*, EURASIP Journal on Advances in Signal Processing **2017** (2017), no. 1, 71.

[86] Ranjeet Kumar Tiwari, Shovan Bhaumik, Paresh Date, and Thiagalingam Kirubarajan, *Particle Filter for Randomly Delayed Measurements With Unknown Latency Probability*, Sensors **20** (2020), no. 19.

[87] Ruud van der Pas, Eric Stotzer, and Christian Terboven, *Using OpenMP – The Next Step: Affinity, Accelerators, Tasking, and SIMD*, 1st ed., The MIT Press, 2017.

[88] Peter Jan Van Leeuwen, Hans Künsch, Lars Nerger, Roland Potthast, and Sebastian Reich, *Particle Filters for High-Dimensional Geoscience Applications: A Review*, Quarterly Journal of the Royal Meteorological Society **145** (2019).

[89] A. Varsi, L. Kekempanos, J. Thiyagalingam, and S. Maskell, *Parallelising Particle Filters With Deterministic Runtime on Distributed Memory Systems*, IET Conference Proceedings (2017), 11–18 (English).

[90] ———, *A Single SMC Sampler on MPI That Outperforms a Single MCMC Sampler"*, eprint arXiv:1905.10252 (2019).

[91] A. Varsi, J. Taylor, L. Kekempanos, E. Pyzer Knapp, and S. Maskell, *A Fast Parallel Particle Filter for Shared Memory Systems*, IEEE Signal Processing Letters **27** (2020), 1570–1574.

[92] Alessandro Varsi and Simon Maskell, *Method of parallel implementation in distributed memory architectures*, GB Patent Request 2101274.5, Jan 29, 2021.

[93] F. Wang and M. Lu, *Efficient Visual Tracking via Hamiltonian Monte Carlo Markov Chain*, The Computer Journal **56** (2013), no. 9, 1102–1112.

[94] T. Wang, Z. Liu, and N. Mrad, *A Probabilistic Framework for Remaining Useful Life Prediction of Bearings*, IEEE Transactions on Instrumentation and Measurement (2020), 1–1.

[95] Ziqi Wang, Marco Broccardo, and Junho Song, *Hamiltonian Monte Carlo Methods for Subset Simulation in Reliability Analysis*, Structural Safety **76** (2019), 51 – 67.

[96] S. White, N. Verosky, and T. Newhall, *A CUDA-MPI Hybrid Bitonic Sorting Algorithm for GPU Clusters*, 2012 41st International Conference on Parallel Processing Workshops, Sep. 2012, pp. 588–589.

[97] Z. Wu and S. Li, *Reliability Evaluation and Sensitivity Analysis to AC/UHVDC Systems Based on Sequential Monte Carlo Simulation*, IEEE Transactions on Power Systems **34** (2019), no. 4, 3156–3167.

[98] Peter Young, *Physics 115/242 the Leapfrog Method and Other "Symplectic" Algorithms for Integrating Newton's Laws of Motion*, (2013).

[99] Jungen Zhang and Hongbing Ji, *Distributed Multi-Sensor Particle Filter for Bearings-Only Tracking*, International Journal of Electronics - INT J ELECTRON **99** (2012), 239–254.

[100] Ran Zhu, Yunli Long, Yaoyuan Zeng, and Wei An, *Parallel Particle PHD Filter Implemented on Multicore and Cluster Systems*, Signal Processing **127** (2016), no. C, 206–216.

[101] E. Özkan, F. Lindsten, C. Fritsche, and F. Gustafsson, *Recursive Maximum Likelihood Identification of Jump Markov Nonlinear Systems*, IEEE Transactions on Signal Processing **63** (2015), no. 3, 754–765.