A SINGLE SMC SAMPLER ON MPI THAT OUTPERFORMS A SINGLE MCMC SAMPLER

Alessandro Varsi
Department of Electrical Engineering and Electronics
University of Liverpool
Liverpool, L69 3GJ, UK
Alessando.Varsi@liverpool.ac.uk

Lykourgos Kekempanos
Department of Electrical Engineering and Electronics
University of Liverpool
Liverpool, L69 3GJ, UK
L.Kekempanos@liverpool.ac.uk

Jeyarajan Thiyagalingam
Scientific Computing Department
Rutherford Appleton Laboratory, STFC
Didcot, Oxfordshire, OX11 0QX, UK
t.jeyan@stfc.ac.uk

Simon Maskell
Department of Electrical Engineering and Electronics
University of Liverpool
Liverpool, L69 3GJ, UK
S.Maskell@liverpool.ac.uk

ABSTRACT

Markov Chain Monte Carlo (MCMC) is a well-established family of algorithms which are primarily used in Bayesian statistics to sample from a target distribution when direct sampling is challenging. Single instances of MCMC methods are widely considered hard to parallelise in a problem-agnostic fashion and hence, unsuitable to meet both constraints of high accuracy and high throughput. Sequential Monte Carlo (SMC) Samplers can address the same problem, but are parallelisable: they share with Particle Filters the same key tasks and bottleneck. Although a rich literature already exists on MCMC methods, SMC Samplers are relatively underexplored, such that no parallel implementation is currently available. In this paper, we first propose a parallel MPI version of the SMC Sampler, including an optimised implementation of the bottleneck, and then compare it with single-core Metropolis-Hastings. The goal is to show that SMC Samplers may be a promising alternative to MCMC methods with high potential for future improvements. We demonstrate that a basic SMC Sampler with 512 cores is up to 85 times faster or up to 8 times more accurate than Metropolis-Hastings.

***K*eywords** Distributed memory architectures · Metropolis-Hastings · Message Passing Interface · Parallel SMC Samplers · Particle Filters.

1 Introduction

In Bayesian statistics, it is often necessary to collect and compute random samples from a probability distribution. Markov Chain Monte Carlo (MCMC) methods are commonly used to address this problem since direct sampling is often hard or impossible. Sequential Monte Carlo (SMC) Samplers are a member of the broader class of SMC methods (which also includes Particle Filters) and can be used in the same application domains as MCMC [1]. While many papers on Particle Filters or MCMC methods exist, SMC Samplers still remain relatively unexplored as a replacement to MCMC.

Research has been focused on improving the run-time and accuracy of SMC and MCMC methods to meet the constraints of modern applications. While accuracy has been improved by several approaches ranging from using better proposal distributions [2] to better resampling and better recycling [3], to improve the run-time these algorithms need to employ parallel computing.

Generic MCMC methods are not parallelisable by nature as it is hard for a single Markov chain to be processed simultaneously by multiple processing elements. In [4], an approach which aims to parallelise a single chain is presented but it quickly becomes problem-specific because the efficiency of parallelisation is not guaranteed, especially for computationally cheap proposal distributions. We acknowledge that one could implement multiple instances of

MCMC in parallel as in [5], but argue that we could also apply the same idea to multiple instances of SMC samplers. However, all chains also need to burn-in concurrently, making it difficult to use this approach to reduce the run-time. In this paper, we seek to develop a parallel implementation of a single instance of a sampling algorithm which outperforms a single MCMC algorithm both in terms of run-time and accuracy. Therefore, we leave comparisons with multiple-chain MCMC to future work along with comparisons with parallel instances of SMC Samplers.
Particle Filters offer inherent parallelism, although an efficient parallelisation is not trivially achievable. The resampling step, which is necessary to respond to particle degeneracy [6], is indeed a challenging task to parallelise. This is due to the problems encountered in parallelising the constituent redistribute step. Initial approaches to performing resampling are explained in [6] [7] and achieve $O(N)$ time complexity. In [8], it has been proven that redistribute can be parallelised by using a divide-and-conquer approach with time complexity equal to $O((log_2 N)^3)$. This algorithm has been optimised and implemented on MapReduce in [9] and then ported to Message Passing Interface (MPI) in [10]. Although the time complexity is improved to $O((log_2 N)^2)$, it has been shown that at least 64 parallel cores are required to outperform the $O(N)$ redistribute version when all other steps are parallelised using MPI.
No parallel implementation of the SMC Sampler on MPI is currently available, despite its similarities with the Particle Filter. Hence, the first goal of this paper is to show that an MPI implementation of the SMC Sampler can be translated from the MPI Particle Filter in [10] by porting its key components. An optimisation of the redistribute in [10] will also be discussed and included in the proposed algorithm. This paper also compares, both in terms of run-time and accuracy, a basic implementation of the SMC Sampler on MPI with an equally simple MCMC method, Metropolis-Hastings [11]. By proving that the SMC Sampler can outperform at least one instance of MCMC, the goal is to clear the way for future research (which space constraints prohibit exploring extensively herein). That future research can then optimise the SMC Sampler and compare it with better-performing MCMC methods, such as TMCMC [12] or HMC [13], in the context of both single and multiple chains (see above). In doing so, optimisations of SMC Samplers may include improved L-kernels, proposal distributions and a full comparison of resampling implementations (akin to that done in [14] in the context of a single core).
The rest of the paper is organised as follows: in Section 2 we give some information about distributed memory architectures and MPI. In Section 3, we describe Metropolis-Hastings and SMC methods with a focus on similarities and differences between Particle Filters and SMC Samplers. In Section 4, we introduce our novel implementation strategy. In Section 5, we describe and show the results of several exemplary case studies with a view to showing worst-case performance, maximum speed-up and space complexity of our MPI implementation of the SMC Sampler and its performance versus Metropolis-Hastings. In Section 6, we draw our conclusions and give suggestions for future improvements.

2 Distributed Memory Model

Distributed memory architectures are a type of parallel system which are inherently different from shared memory architectures. 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.
The main advantages relative to shared memory architectures include scalable memory and computation capability with the number of cores and a guarantee of there being no interference when a core accesses its own memory. The main disadvantage is the cost of communication and consequent data movement. This may affect the speed-up relative to a single-core.
In order to implement the algorithms we discuss in this paper, we use Message Passing Interface (MPI) which is one of the most common programming models for distributed memory environments. In this model, the cores are uniquely identified by a rank, connected via communicators and they use Send/Receive communication routines to exchange messages.

3 SMC and MCMC methods

In this section, we provide details about MCMC and SMC methods with a view to showing similarities and differences between Particle Filters and SMC Samplers. The reader is referred to [1], [6] and [11] for further details.

3.1 Sequential Monte Carlo methods

SMC methods apply the Importance Sampling principle to make Bayesian inferences. The main idea consists of generating N statistically independent hypotheses called particles (or samples) at every given iteration t. The population of particles $\mathbf{x}_t \in \mathbb{R}^{N \times M}$ is sampled from a user-defined proposal distribution $q(\mathbf{x}_t|\mathbf{x}_{t-1})$ such that $\mathbf{x}_t$ represents the pdf of the state of a dynamic model (in Particle Filters) or samples from a static target posterior distribution (in SMC

Samplers[1]). Each particle $\mathbf{x}_t^i$ is then assigned to an unnormalised importance weight $w_t^i = F(w_{t-1}^i, \mathbf{x}_t^i, \mathbf{x}_{t-1}^i)$, such that the array of weights $\mathbf{w}_t \in \mathbb{R}^N$ provides information on which particle best describes the real state of interest.
The particles are however subjected to a phenomenon called degeneracy which (within a few iterations) makes all weights but one decrease towards 0. This is because the variance of the weights is proven to increase at every iteration [6]. There exist different strategies to tackle degeneracy. The most common is to perform a resampling step which repopulates the particles by eliminating the most negligible ones and duplicating the most important ones. Different variants of resampling exist [14] and the chosen methodology is described in detail in Section 3.3. Resampling is only triggered when it is needed, more precisely when the (approximate) effective sample size

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

decreases below a certain threshold N^* (which is usually set to $\frac{N}{2}$). $\tilde{\mathbf{w}}_t \in \mathbb{R}^N$ represents the array of the normalised weights, each of them calculated as follows:

$$\tilde{w}_t^i = \frac{w_t^i}{\sum_{j=0}^{N-1} w_t^j} \tag{2}$$

At every iteration, estimates are produced as a weighted sum of $\mathbf{x}_t$, weighted using $\tilde{\mathbf{w}}_t$.

3.1.1 Particle Filters

A range of different Particle Filter methods exist. This section provides a brief description of Sequential Importance Resampling (SIR), described by Algorithm 1 in the appendix.
Let $\mathbf{X}_t \in \mathbb{R}^M$ be the current state of the dynamic system that we want to estimate. At every time step t a new measurement $\mathbf{Y}_t \in \mathbb{R}^D$ is collected. In the SIR Filter, the weighted particles are initially drawn from the prior distribution $q(\mathbf{x}_0) = p(\mathbf{x}_0)$ and then drawn from the proposal distribution as follows:

$$\mathbf{x}_t^i \sim q(\mathbf{x}_t^i | \mathbf{x}_{t-1}^i, \mathbf{Y}_t) \tag{3}$$

The weights are initially set to $1/N$ and then computed as

$$w_t^i = F(w_{t-1}^i, \mathbf{x}_t^i, \mathbf{x}_{t-1}^i) = w_{t-1}^i \frac{p(\mathbf{x}_t^i | \mathbf{x}_{t-1}^i) p(\mathbf{Y}_t | \mathbf{x}_t^i)}{q(\mathbf{x}_t^i | \mathbf{x}_{t-1}^i, \mathbf{Y}_t)} \tag{4}$$

The weights are then normalised and used to calculate N_{eff} as in (1). Then resampling is performed if needed. In the last step, the estimation of the state is given by the weighted mean of the particles.

3.1.2 SMC Samplers with recycling

Like MCMC methods, the goal in the SMC Samplers is to draw samples from a static target distribution of interest $\pi_t(\mathbf{x}_t)$. The algorithm begins by drawing N samples from the initial proposal $q(\mathbf{x}_0)$ and giving the i-th sample the weight $w_0^i = \frac{\pi_0(\mathbf{x}_0^i)}{q_0(\mathbf{x}_0^i)}$.
After the first iteration, the samples are drawn from the forward Markov kernel, $q_t(\mathbf{x}_t | \mathbf{x}_{t-1})$, while the weights require backward Markov kernels $L_t(\mathbf{x}_{t-1} | \mathbf{x}_t)$ as follows:

$$w_t^i = F(w_{t-1}^i, \mathbf{x}_t^i, \mathbf{x}_{t-1}^i) = w_{t-1}^i \frac{\pi_t(\mathbf{x}_t^i)}{\pi_t(\mathbf{x}_{t-1}^i)} \frac{L_t(\mathbf{x}_{t-1}^i | \mathbf{x}_t^i)}{q_t(\mathbf{x}_t^i | \mathbf{x}_{t-1}^i)} \tag{5}$$

As is the case for Particle Filters, after the importance weights evaluation and normalisation, the resampling step may be triggered depending on the value of N_{eff}.
In the vanilla SMC Sampler, estimates are performed according to the particles in the final iteration. The expected value is computed by multiplication of the particles at the final iteration T with the corresponding weights. In [3], a novel recycling method is proposed. Instead of considering the particles from the last iteration as providing the outputs, estimates are computed using all particles from all iterations. Using the notation of this paper, estimates are performed as follows:

$$\hat{\mathbf{f}} = \frac{\sum_{t=1}^{T} \mathbf{f}_t \tilde{c}_t}{\sum_{t=1}^{T} \tilde{c}_t} \tag{6}$$

[1]While it is not discussed here extensively, SMC Samplers can also be configured to offer improved performance in contexts where a Particle Filter struggles [15].

where $\mathbf{f}_t$ is calculated as

$$\mathbf{f}_t = \sum_{i=0}^{N-1} \mathbf{x}_t^i \tilde{w}_t^i \tag{7}$$

and the normalisation constant[2] is

$$\tilde{c}_t = \int \pi(\mathbf{x}_t) d\mathbf{x}_t \approx c_t = \frac{\sum_{i=0}^{N-1} w_t^i}{\sum_{i=0}^{N-1} w_{t-1}^i} \tag{8}$$

Algorithm 2 in the appendix describes the SMC Sampler with the recycling method.

3.2 Metropolis-Hastings Algorithm

The Metropolis-Hastings (MH) algorithm (see Algorithm 3 in the appendix) simulates a Markov chain where, at each iteration, a new sample, $\mathbf{x}^*$, is drawn from a proposal distribution. The new sample is accepted or rejected using the Rejection Sampling principle with acceptance probability $a = min\{1, \frac{\pi(\mathbf{x}^*)q(\mathbf{x}|\mathbf{x}^*)}{\pi(\mathbf{x})q(\mathbf{x}^*|\mathbf{x})}\}$. To reduce the dependency on the initial sample, the first (user-defined) τ samples are discharged (burn-in).

3.3 Key components of Particle Filters and SMC Samplers

Algorithms 1 and 2 in the appendix show that SIR Particle Filter and SMC Sampler with recycling share the same key components.

Importance Sampling is trivially parallelisable as (3), (4) and (5) are element-wise operations. Hence, Importance Sampling achieves $O(1)$ time complexity for $P = N$ cores.

Expressions (1), (2), (7), and the weighted mean of particles require Sum and then can be easily parallelised by using Reduction. The time complexity of any Reduction operation scales logarithmically with the number of cores.

Both algorithms invoke resampling if $N_{eff} < N^*$. Several alternative resampling steps have been proposed and a comparison between them is discussed in [14]. These algorithms solve the problem in $O(N)$ operations. The key idea of these algorithms is to process $\mathbf{w}_t$ to generate an array of integers called $\mathbf{ncopies} \in \mathbb{Z}^N$ whose i-th element, $ncopies^i$, indicates how many times the i-th particle has to be duplicated. It is easy to infer that $\mathbf{ncopies}$ has the following property:

$$\sum_{i=0}^{N-1} ncopies^i = N \tag{9}$$

In previous work to parallelise Particle Filters described in [6], [8], [9] and [10], Minimum Variance Resampling (MVR), a variant of Systematic Resampling in [14], has always been the preferred resampling scheme. Since this paper is built on the results in [10], MVR will be the only variant of resampling we consider. This algorithm uses Cumulative Sum to calculate the CDF and then it generates $\mathbf{ncopies}$ such that the new population of particles has minimum ergodic variance. After that, it is necessary to perform a task called redistribute which duplicates $\mathbf{x}_t^i$ as many times as $ncopies^i$. This task has already been identified as bottleneck (see [6], [8], [9]) and it will be discussed in detail in the next section. We note that the reset step (which sets all the weights to $1/N$) after redistribute is trivially parallelised.

3.3.1 Redistribute

The redistribute step is necessary to regenerate the population of particles and is a task which all resampling variants have in common. A naive and mature implementation can be found in [6] [7]. The same is described by Algorithm 5 in the appendix and referred to as Sequential Redistribute (S-R) in the rest of the paper. This routine simply iterates over $\mathbf{ncopies}$ and, for the $j-$th element, it copies $\mathbf{x}^j$ as many times as $ncopies^j$. Considering that $\mathbf{ncopies}$ follows (9), it is easy to infer that S-R achieves $O(N)$ time complexity with a very low constant time. However, this algorithm is not trivial to parallelise because the workload among the processors cannot be readily distributed deterministically. This is because $ncopies^j$ could be equal to any value between 0 and N. Parallelisation is even more complicated on distributed memory architectures since one core cannot directly access the memory of the other cores [10].

In [8], it has been shown that, by using a top-down divide-and-conquer approach, redistribute can be parallelised. Starting from the root node, the key idea consists of sorting $\mathbf{ncopies}$ and moving the particles at every stage of a binary tree. This can be achieved by searching for a particular index called $pivot$ which perfectly divides the node into two balanced leaves. Once $pivot$ is identified, the node is split. In order to find $pivot$, Cumulative Sum (whose parallel implementation runs in $O(log_2 N)$ steps [16]) is performed and then $pivot$ is the first index where Cumulative Sum is equal to or greater than $\frac{N}{2}$. This routine is repeated recursively $log_2 N$ times. Since Bitonic Sort is the chosen parallel sorting algorithm and it is known that its time complexity is equal to $O((log_2 N)^2)$ with $P = N$ cores, then we can infer that this redistribute achieves $O((log_2 N)^3)$ time complexity for the same level of parallelism. Sorting the particles is required to make sure that the splitting phase can be performed deterministically in $O(1)$.

[2] (8) is equivalent to (14) in [3], albeit with simplified notation here.

In [9], the redistribute algorithm in [8] was improved by making a subtle consideration: the workload can still be divided deterministically if we perform Bitonic Sort only once. After this single sort, the algorithm moves on to another top-down routine where we use rotational shifts to shift all particles on the left side of $pivot$ up to the left side of the node. This way the father node gets split into two balanced leaves. This algorithm is recursively performed $O(log_2 N)$ times until the workload is equally distributed across the cores; then S-R is called. Algorithm 7 in the appendix summarises this routine and, in this paper, is described as Bitonic Sort Based Redistribute (B-R). Rotational shifts are faster than Bitonic Sort as the achieved time complexity is equal to $O(log_2 N)$ and, therefore, the overall time complexity is improved to $O((log_2 N)^2)$. In [9], B-R has been implemented on MapReduce and, although it was significantly better than the algorithm in [8], its runtime for 512 cores was up to 20 times worse than a single-core S-R. In [10], B-R has been ported to distributed memory architectures by using MPI and compared to a deterministic MPI implementation of S-R, in which one core gathers all particles from the other cores, performs S-R locally and scatters back the resulting array. To avoid misunderstanding, we refer to the MPI implementation of S-R in [10] as Centralised Redistribute (C-R), which is described by Algorithm 6 in the appendix. The results indicate that the scalability of the MPI implementation is improved relative to the scalability achieved using MapReduce because B-R on MPI could outperform C-R for at least $P = 64$ cores. Possible ways to improve B-R are discussed in the next session.

4 Novel Implementation

In this section, we consider ways to improve redistribute and how an MPI SMC Sampler could be an alternative to Metropolis-Hastings.

4.1 Improving single-core Bitonic Sort

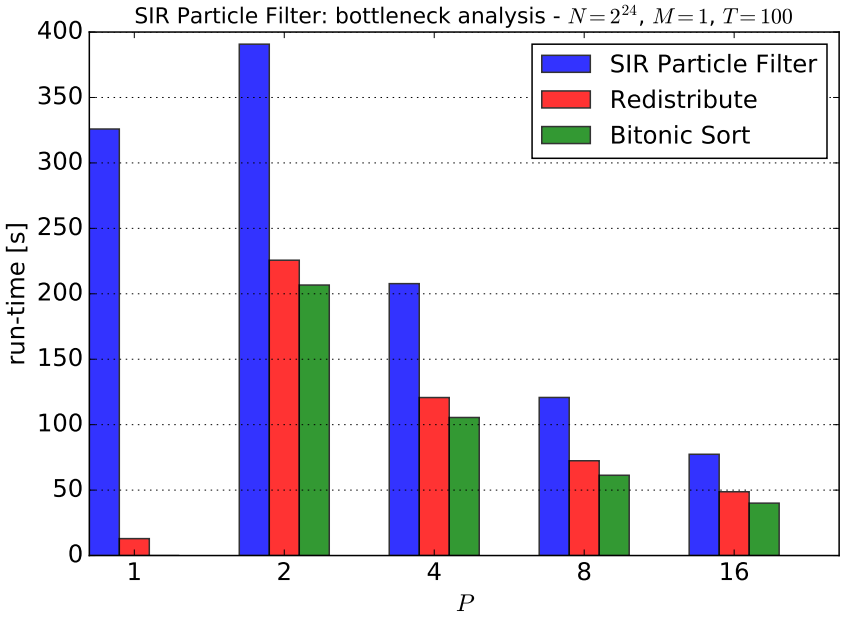


Figure 1: SIR Particle Filter: bottleneck analysis - $N = 2^{24}$, $M = 1$, $T = 100$

As is observed here and has been discussed elsewhere in the literature [9] [10], the redistribute step is the bottleneck that complicates parallel implementation of Particle Filters. To ensure this is clear, we repeat an experiment from [10] and report the results of the same SIR Particle Filter with B-R within using $N = 2^{24}$, $T = 100$ in Figure 1. The run-times vs the number of cores P for the entire Particle Filter, the constituent redistribute step and the subset of redistribute that is taken up with the Bitonic Sort step are given. As we can see, for $P > 1$, redistribute always accounts for at least 50% of the total run-time and this proportion increases with P. For the same values of P, we can also observe that Bitonic Sort is by far the most computationally intensive task within redistribute and hence is the true bottleneck of the Particle Filter.

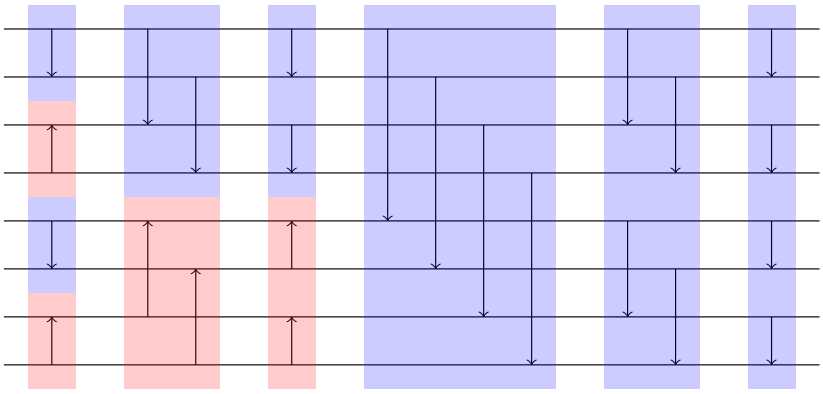


Figure 2: Bitonic Sort & Nearly Sort - Sorting Network

Bitonic Sort is a very fast comparison-based parallel sorting algorithm [17]. This algorithm uses a divide-and-conquer approach to first divide the input sequence into a series of Bitonic sequences[3]. 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 2. 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 application of Bitonic Merge (blue or red if Merge is called in increasing or decreasing order respectively).
It has been proven that, given a generic array of N elements, Bitonic Sort solves the problem in $O(N(log_2 N)^2)$ comparisons [17]. Bitonic Sort is, however, suitable to run in parallel by making P cores work on different chunks of the input array. In this case, each wire (or groups of wires) in Figure 2 may also represent a core (or the elements that each core operates on). When $P = N$, it is easy to infer that the achieved time complexity is equal to $O((log_2 N)^2)$. More generically, we can say that for any number of cores $P \leq N$ the time complexity is equal to

$$O\left(\frac{N}{P}\left(log_2\left(\frac{N}{P}\right)\right)^2 + \frac{N}{P}\left(log_2 P\right)^2\right) \tag{10}$$

$\frac{N}{P}(log_2(\frac{N}{P}))^2$ is the time complexity to perform Bitonic Sort locally before the cores start interacting with each other. This term is definitely dominant, especially for low values of P. 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, as was been suggested in [10].
In the literature, there are plenty of alternatives to Bitonic Sort available. Algorithms such as Quicksort, Mergesort and Heapsort, for example, achieve $O(Nlog_2 N)$ 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 (9) 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 $O(Nlog_2 N)$ 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 which we consider in the experiment in Section 5.1. A Bitonic Sorter with Mergesort performed locally achieves the following time complexity:

$$O\left(\frac{N}{P}log_2\left(\frac{N}{P}\right) + \frac{N}{P}(log_2 P)^2\right) \tag{11}$$

We also observe that **ncopies** is an array of integers. Hence, one could locally use linear time sorting algorithms such as Counting Sort or Radix Sort (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 could be very high, the temporary array may not fit within the local memory of a single machine. This problem

[3] 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.

is shared with C-R and the impact of this issue will be discussed in Section 5.4. 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, we are looking for 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 a core achieves these aims, in the next two sessions, we go on to develop an improved strategy that is sufficient for our needs and does not require sort at all.

4.2 Nearly Sort: an alternative to single-core sorting

In [6], sorting was used extensively. In B-R, rotational shifts are used $log_2 P$ times while Bitonic Sort is used only once. This replacement of sort with rotational shift 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 we do 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 we first provide the following definition.

Definition 1 *Let* $\mathbf{g}$ *be a sequence of* N *elements.* $\mathbf{g}$ *is called Nearly-Sorted sequence when it has the following shape:* $\left[0, ..., 0, g^0, g^1, ..., g^{m-1}\right]$ *where* $g^i > 0 \; \forall i = 0, 1, ..., m-1$ *and* $0 \leq m \leq N$. $\mathbf{g}$ *is an ascending Nearly-Sorted sequence if the first elements of* $\mathbf{g}$ *are* 0 *and a descending Nearly-Sorted sequence if the final elements are* 0.

We can infer that the workload can be divided deterministically if $\mathbf{ncopies}$ is a Nearly-Sorted sequence. In B-R, this condition is ensured by sorting before the subsequent parts of the redistribute step. While there are single-core sorting algorithms that achieve $O(N)$ time complexity, these algorithms do not satisfy our need for 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.
Algorithm 8 in the appendix, which we have called Sequential Nearly Sort (S-NS), declares two iterators l and r which respectively point at the first and the last element of $\mathbf{ncopies}$. Step by step, the i-th element of $\mathbf{ncopies}$ is considered and if the value is higher than 0 then the particle is copied to the right end of the output array. If not, it gets copied to the left end. The output $\mathbf{ncopies}_{new}$ will then be an ascending Nearly-Sorted sequence. S-NS requires N iterations of the for loop, which means that it achieves $O(N)$ time complexity or $O(\frac{N}{P})$ if we consider that each core owns $\frac{N}{P}$ 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.

4.3 Parallel Nearly Sort

We want S-NS to be used as part of a parallel algorithm which generates a Nearly-Sorted sequence from a random input one. We now discuss how to achieve this.

Definition 2 *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.
We need to adapt Bitonic Merge such that it processes a Nearly-Bitonic sequence and returns a monotonic Nearly-Sorted sequence. We call this algorithm 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 call MPI_Sendrecv to 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 2), 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.
Figure 3 illustrates a possible example of Nearly Merge where each core owns 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. We can infer that the overall time complexity for Nearly Sort is equal to

$$O\left(\frac{N}{P} + \frac{N}{P}(log_2 P)^2\right) \tag{12}$$

This algorithm 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 expect this algorithm to outperform both

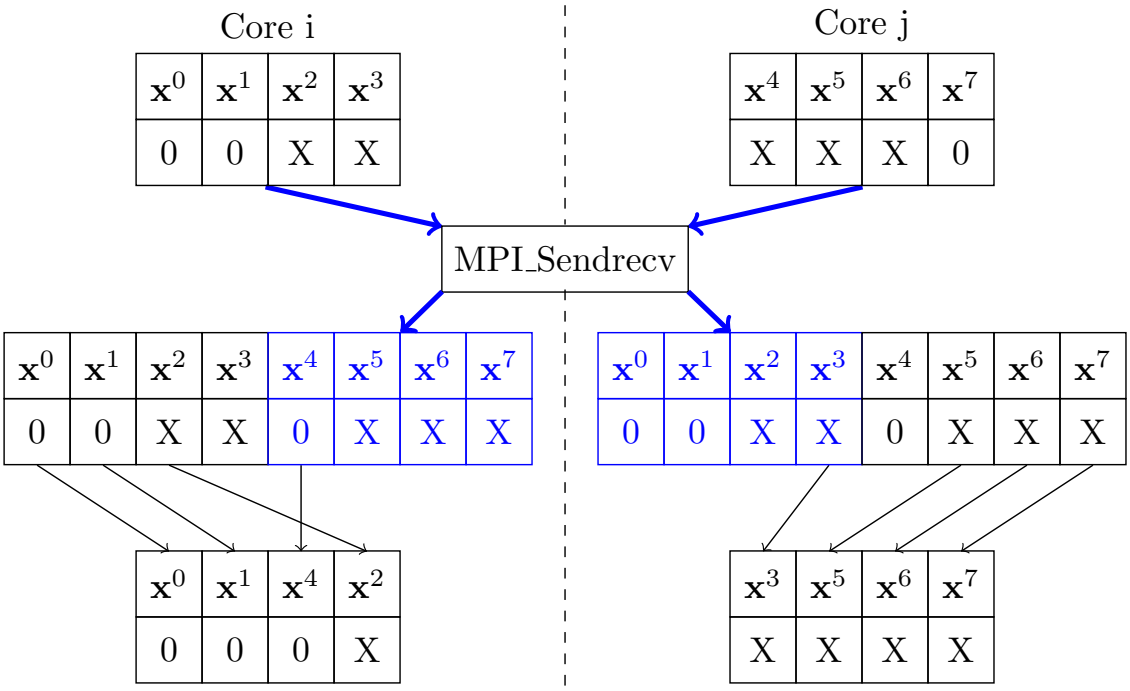


Figure 3: Nearly Merge - example

Bitonic Sort and a Bitonic Sorter with Mergesort performed locally. By extension, if we exchange Bitonic Sort with Nearly Sort in B-R we expect to have better performance. Algorithm 9 in the appendix describes the new routine. A possible example for $N = 16$ and $P = 4$ is shown in Figure 4. From now on, we refer to this algorithm as Nearly Sort Based Redistribute (N-R).
In SMC methods **ncopies** is the array to (nearly) sort and each key $ncopies^i$ is necessarily coupled to the particle $\mathbf{x}^i \in \mathbb{R}^M$. (12) must then be extended to:

$$O\left(\frac{M \cdot N}{P} + \frac{M \cdot N}{P}(log_2 P)^2\right) \tag{13}$$

We denote that (13) can qualitatively describe the time complexity of N-R and, by extension, the time complexity of an SMC method which uses N-R within. The same conclusions about (12) and (13) can be made about (10) and (11) but they are left out for brevity.

4.4 Single SMC Sampler vs Single Metropolis-Hastings

Metropolis-Hastings and the SMC Sampler perform sampling from a target distribution and they both provide an accurate result for a sufficiently large number of iterations. However, the details of the two approaches differ substantially. As we have discussed in Section 3, Metropolis-Hastings uses a Markov Chain to generate each sample one by one based on the history of the previous samples. This approach makes a single instance of Metropolis-Hastings hard to parallelise in a problem-agnostic way. On the other hand, the SMC Sampler is a population-based algorithm where all samples are processed independently and concurrently during each iteration.
Now let the total simulation-time for each algorithm be fixed to Δ seconds. After Δ seconds, Metropolis-Hastings and the SMC Sampler will have performed T_{MH} and T_{SMC} iterations respectively and provide a solution with a certain root mean squared error. Since a single Metropolis-Hastings is hard to parallelise, we cannot increase its accuracy without running the simulation for longer than Δ seconds. However, a single SMC Sampler can improve its throughput or accuracy by taking advantage of its inherent parallelism. In an ideal world, 2 cores can perform T_{SMC} iterations in $\frac{\Delta}{2}$ seconds, but they can also, and most importantly, run $2T_{SMC}$ iterations in Δ seconds. This means they can achieve better accuracy with the same run-time. By extension, P cores can ideally run $P \cdot T_{SMC}$ iterations in Δ seconds but they will achieve a much better accuracy than a single core is capable of.
The main goal of this paper is to prove that a P core MPI SMC Sampler can be more accurate over the same run-time than Metropolis-Hastings when they sample from the same target distribution and use the same proposal. A more exhaustive explanation with experimental results is provided in Section 5.5.1.

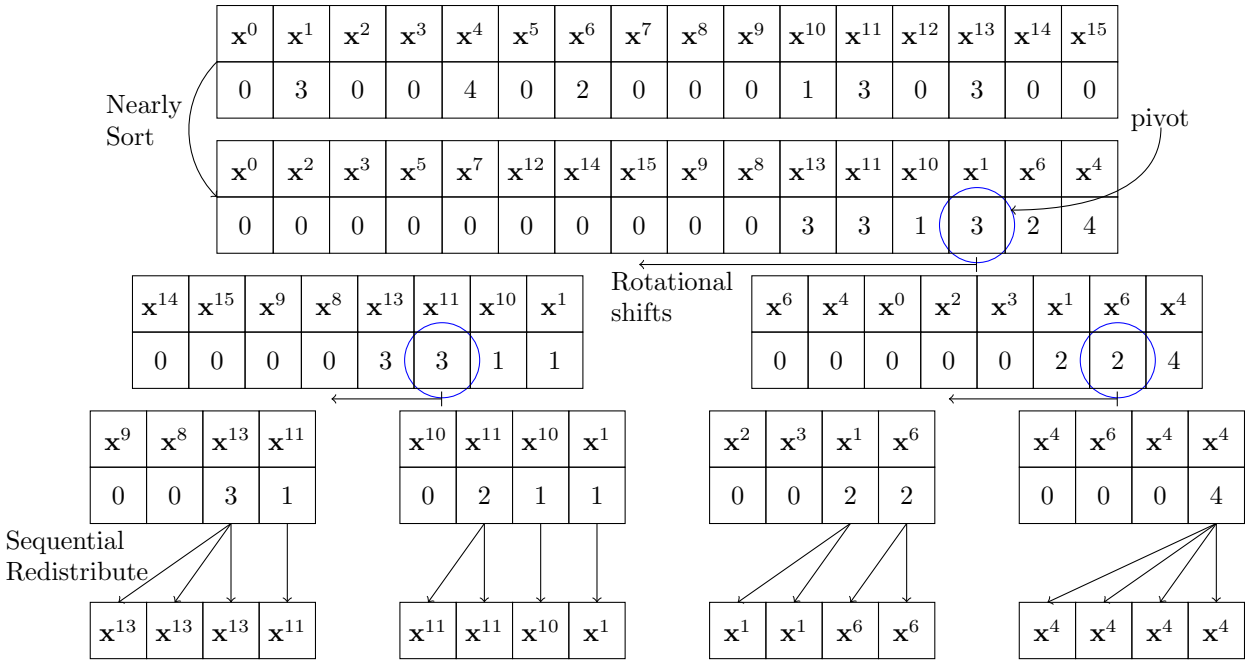


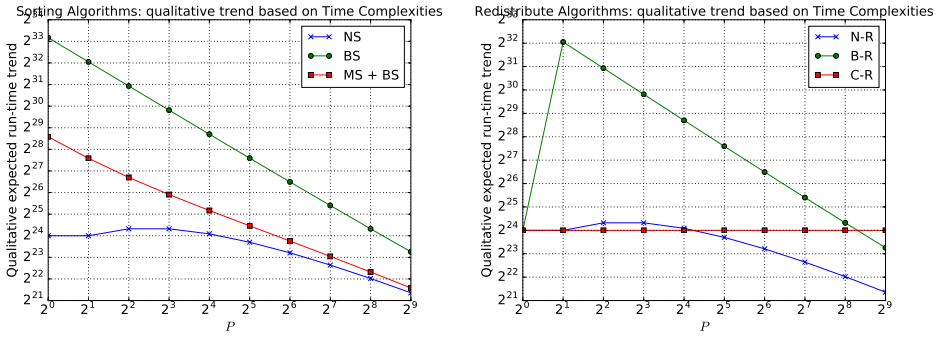
Figure 4: Nearly Sort Based Redistribute

5 Case Studies and results

In this section, we briefly describe the experiments we make and we analyse the results. Table 1 provides details about Barkla and Chadwick, the two platforms we use for the described experiments. Barkla is the preferred cluster for the majority of the experiments as it can provide more resources.

5.1 Bottleneck

To evaluate the improvements in the bottleneck, we first focus on the sorting phase. Then we compare N-R, B-R and C-R. $M = 1$ in this first experiment for brevity.



(a) Sorting

(b) Redistribute

Figure 5: Bottleneck: theoretical run-time trend

5.1.1 Sorting

As we outlined in Section 4, Bitonic Sort is the slowest task in B-R. In this experiment, we compare three different deterministic sorting/nearly sorting algorithms: Bitonic Sort (BS), Bitonic Sort with Mergesort performed locally (BS+MS) and Nearly Sort (NS). These algorithms are compared by passing in input the same two arrays: **ncopies** and **x**. **ncopies** represents the array of the numbers of copies and hence it is an array of integers. It is generated randomly according to (9) by using a Gaussian random generator followed by MVR. **x** represents an array of single-dimension particles and hence it is an array of floating point numbers which are generated by a Gaussian random generator.

The experiments have been run on Barkla for $N = 2^{10}, 2^{17}, 2^{24}, 2^{31}$ particles and increasing numbers of cores $P = 1, 2, 4, 8, ..., 512$. Both N and P must necessarily be equal to power of 2 numbers, due to the constraint of Bitonic Sort and Nearly Sort. Each experiment has been run 20 times and we report the median of the sampled run-times vs the number of cores.
As we can see in Figure 6, NS does not scale for up to 8 cores. This trend might seem odd but it can be explained by analysing the time complexity of NS described by (12). Figure 5a describes the qualitative trend of (10), (11) and (12). When $P = 1$, the quasilinear term in (12) is equal to 0. However, for $2 \leq P \leq 8$, the quasilinear term offsets the improvement associated with the linear term. Theoretically, the run-time should have positive speed-ups for $P = 32$. In the measured values for $N \geq 2^{17}$, this happens when $P \geq 32$ or $P \geq 64$ cores, depending on N. We associate this slight discrepancy to the additional communication cost associated with larger numbers of particles.
However, the most important result is that NS is significantly faster than the other algorithms and especially BS for a low number of cores. Then, when P increases the performance of both algorithms become closer because the time complexity of both algorithms is asymptotically equal to $O((log_2 N)^2)$, as underlined in Section 4. These results suggest that using Bitonic Sort or Nearly Sort results in similar run-time for $P \geq 512$ but, using Nearly Sort may lead to significant improvements for $P < 512$. This means that the crossing point with respect to the run-time of C-R may be shifted to the left side of the graph, relative to the results in [10].
The results for $N = 2^{10}$ keys show that BS and BS+MS stop scaling for a very low value of P. The reasons behind this result have required further investigation. For a very low number of keys, the granularity of the pipeline is already fine. In other words, the computation cost is already comparable with the communication cost and using more cores does not provide any scalability. NS is also affected by the same problem. In addition, the time complexity of Nearly Sort is necessarily higher than $O(N)$ for $P \leq 8$ cores. For these two reasons NS always returns negative speed-ups.

5.1.2 B-R vs N-R vs C-R

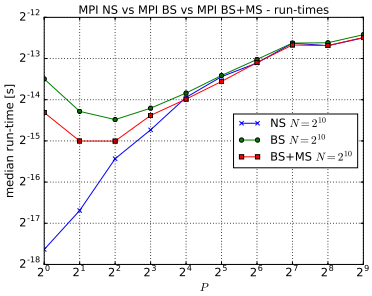
In this experiment, we use exactly the same strategy described in the previous section, since the required input for N-R, B-R and C-R is the same. The results are shown in Figure 6. The results for redistribute with BS+MS are left out for brevity.
As we expected from theory, for $N \geq 2^{17}$ N-R is better than B-R overall and much faster for a small number of cores. However, the most important result is that N-R outperforms C-R at the theoretical minimum (which is $P = 32$) for some values of the dataset size N. This suggests that, as long as we use a parallel redistribute whose time complexity is equal to $O((log_2 N)^2)$, we cannot outperform C-R for $P < 32$ nor have positive speed-up for the same values of P. In order to achieve this goal, a new algorithm with $O(log_2 N)$ time complexity is needed. Sorting networks which achieve the theoretical lower bound have been proposed in [18], [19] which improve the original AKS sorting network presented in [20]. These networks can also be rearranged to perform redistribute by substituting the comparators with balancers. However, they cannot be practically used because each atomic step requires a huge constant time C. The exact value of C is unknown as it also depends on the network parameters but it seems to be in the order of thousands (e.g. $C = 6100$ in the best configuration in [18]). It can then be inferred that they cannot outperform $O((log_2 N)^2)$ sorting networks such as Bitonic Sort for any practical N. In [21] it has been estimated that a hypothetical $C = 87$ would require $N \geq 2^{173}$ to make AKS-like sorting networks faster than Bitonic Sort. Therefore, the infeasibility of this class of algorithms makes $O((log_2 N)^2)$ redistribute on distributed memory systems a practical lower bound (although it cannot yet be considered a theoretical minimum).
For $N = 2^{10}$, N-R does not scale and does not outperform C-R either. As we outlined in the previous section, for low values of N the granularity is already too fine to observe any speed-up.

5.2 Worst-case speed-up

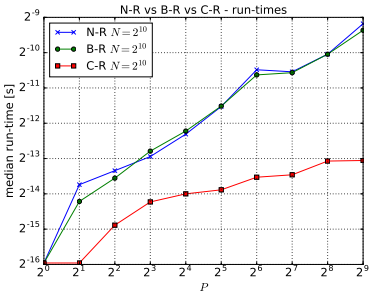
In this section, we analyse three case studies, two for the Particle Filter and one example of an SMC Sampler. Depending on the chosen redistribute (N-R, B-R or C-R), we use the acronyms N-PF/B-PF/C-PF for the Particle Filter, and N-SMCS/B-SMCS/C-SMCS for the SMC Sampler. We consider a worst-case scenario which occurs when resampling is needed at every iteration and the time taken to perform Importance Sampling is small relative to the time taken to perform resampling. This section aims to achieve two goals. The first one is to demonstrate that the historic progress made in developing parallel implementations of Particle Filters can be translated to develop a parallel implementation of an SMC Sampler. The second one is to estimate the worst-case speed-up of our improved algorithm.
All experiments in this section are run for the same values of N as were considered in the previous section. Each run-time is once again the median of 20 runs, each one representing a simulation of 100 iterations in the worst-case scenario and for increasing number of cores $P = 1, 2, 4, ..., 512$.

5.2.1 Particle Filter on Econometrics

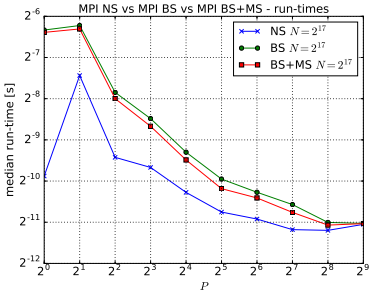
In this experiment, we use Barkla and compare N-PF, B-PF and C-PF. We apply these algorithms to a stochastic volatility model which describes the evolution of the pound-to-dollar exchange rate between the 1st of October 1980 and the 28th of June 1985. This model has been used in [22] to demonstrate the utility of advanced SMC methods, such



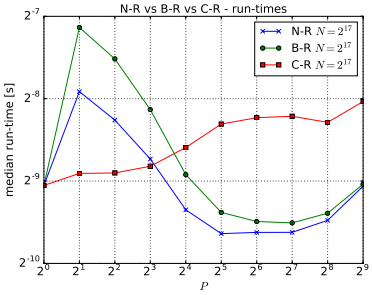
(a) NS, BS and BS+MS for $N = 2^{10}$



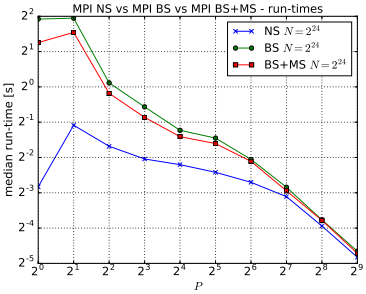
(b) N-R, B-R and C-R for $N = 2^{10}$



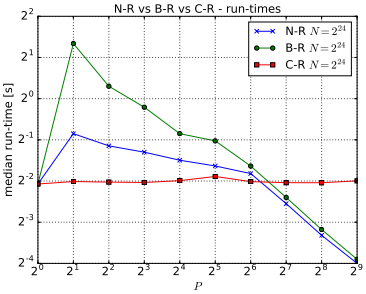
(c) NS, BS and BS+MS for $N = 2^{17}$



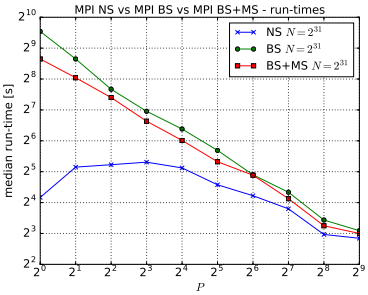
(d) N-R, B-R and C-R for $N = 2^{17}$



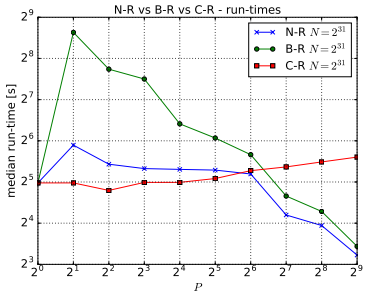
(e) NS, BS and BS+MS for $N = 2^{24}$



(f) N-R, B-R and C-R for $N = 2^{24}$



(g) NS, BS and BS+MS for $N = 2^{31}$



(h) N-R, B-R and C-R for $N = 2^{31}$

Figure 6: Bottleneck: median run-times vs P for $N = 2^{10}, 2^{17}, 2^{24}, 2^{31}$

as Block Sampling Particle Filters, over SIR Particle Filters.

$$X_t = \phi X_{t-1} + \sigma V_t \tag{14a}$$

$$Y_t = \beta \exp\left(\frac{X_t}{2}\right) W_t \tag{14b}$$

(14a) and (14b) represent the model where the coefficients $\phi = 0.9731$, $\sigma = 0.1726$, $\beta = 0.6338$ (as selected in [22]) and the noise terms for the state and the measurement are $V_t \sim \mathcal{N}(0,1)$ and $W_t \sim \mathcal{N}(0,1)$. The initial state is sampled as $X_0 \sim \mathcal{N}(0, \frac{\sigma^2}{1-\phi^2})$. The particles are initially drawn from the prior distribution. The importance weights are simply equal to the likelihood $p\left(Y_t|X_t\right)$ since the dynamic model is used as the proposal.

5.2.2 Particle Filter on Bearing-Only Tracking

This experiment is focused on showing the performance of N-PF, B-PF and C-PF applied to a non-scalar model. The chosen example is the popular four-dimensional state model for a Bearing-Only Tracking problem, where the state is represented by the position and velocity of the tracked object. Both position and velocity are 2-dimensional physical quantities. This model was previously presented in several publications, such as in [7], and used in [22] to test the Block Sampling SIR Filter. In accordance with [22], we consider the state to be composed of four elements denoted such that $\mathbf{X}_t = \left[X_t^0, X_t^1, X_t^2, X_t^3\right]$ where X_t^0, X_t^2 are position and X_t^1, X_t^3 are velocity.
The model is defined as follows:

$$\mathbf{X}_t = \mathbf{A} \cdot \mathbf{X}_{t-1} + \mathbf{V}_t \tag{15a}$$

$$Y_t = \arctan\left(\frac{X_t^2}{X_t^0}\right) + W_t \tag{15b}$$

where the state transition matrix and the covariance are

$$\mathbf{A} = \begin{bmatrix} 1 & \Delta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \Delta \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{\Sigma} = \begin{bmatrix} \frac{5\Delta^3}{3} & \frac{5\Delta^2}{2} & 0 & 0 \\ \frac{5\Delta^2}{2} & 5\Delta & 0 & 0 \\ 0 & 0 & \frac{5\Delta^3}{3} & \frac{5\Delta^2}{2} \\ 0 & 0 & \frac{5\Delta^2}{2} & 5\Delta \end{bmatrix}$$

The noise terms are $\mathbf{V}_t \sim \mathcal{N}\left(\mathbf{0}, \mathbf{\Sigma}\right)$ and $W_t \sim \mathcal{N}\left(0, 10^{-4}\right)$. 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 and it is set to $\Delta = 1$ s.

5.2.3 Sampling using SMC samplers

We apply N-SMCS, B-SMCS and C-SMCS to sample from a static single-dimensional ($M = 1$) Student's t posterior distribution calculated as:

$$\pi(\mathbf{x}) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)\sqrt{(\nu\pi)}}\left(1 + \frac{1}{\nu}(\mathbf{x}-\mu)^2\right)^{-\frac{\nu+1}{2}} \tag{16}$$

where ν and μ correspond to the degrees of freedom and the mean value respectively.
In the experiment, the particles, as samples of $\mathbf{x}_t$, at the t-th iteration are generated using random walk as the proposal distribution, $\mathbf{x}_t \sim \mathcal{N}(\mathbf{x}_{t-1}, \epsilon)$. The backward kernel is naively selected to emulate MCMC such that $L_t(\mathbf{x}_{t-1}|\mathbf{x}_t) = q_t(\mathbf{x}_t|\mathbf{x}_{t-1})$. We then anticipate that a better choice of $L_t(\mathbf{x}_{t-1}|\mathbf{x}_t)$ can positively impact our estimate. The recycling described in Section 3.1.2 is also used.

5.2.4 Worst-case results

In this section we provide the results of the experiments described in Sections 5.2.1, 5.2.2 and 5.2.3 which are shown in Figure 7 for $N = 2^{10}, 2^{17}, 2^{24}, 2^{31}$ respectively. Since that N-R, B-R and C-R have the same baseline, we show the speed-ups instead of the run-times as, in this case, these speed-ups can also prove which algorithm is faster.
As we can see, for $N = 2^{10}$ N-PF/N-SMCS does not outperform C-PF/C-SMCS. The reasons are the granularity and the theoretical time complexity of N-R as we explained in Section 5.1. In contrast, C-PF/C-SMCS can keep scaling for a relatively low P, until redistribute emerges as the bottleneck. However, since modern applications need a large number of particles (2^{10} is just 1024), we are not discouraged by these limitations.
For $N \geq 2^{17}$, computation becomes dominant over communication and both N-PF/N-SMCS and B-PF/B-SMCS can scale for much larger values of P and can both outperform C-PF/C-SMCS. For this range of N, using Nearly Sort instead of Bitonic Sort makes the SMC method faster for any number of cores and up to twice as fast as using Bitonic

Sort for low values of P. The gap between these two approaches also increases with N. N-PF/N-SMCS runs faster than the solution with C-R in the range $32 \leq P \leq 128$ and it can be as much as approximately 3 times faster for $P = 512$ cores.
Overall, we can say that for a fixed N, changing the model in the Particle Filter (i.e. whether we consider stochastic volatility or Bearing-Only Tracking) or switching from Particle Filters to SMC Samplers gives roughly the same trend. This proves that the improvements that have been demonstrated in the context of Particle Filters can directly be translated to the context of SMC Samplers.
As we can see, the minimum worst-case speed-up is about 40 and occurs in the context of the Bearing-Only Tracking. On the other hand, the maximum worst-case speed-up can be up to 100: this occurs in the context of the SMC Sampler. The efficiency of N-SMCS with respect to the maximum speed-up is indeed significantly higher than for the Particle Filter. This is due to the different way particles and weights are calculated in the SMC Sampler. For example, (16) is more computationally intensive than (14a), and the likelihood calculation in the SMC Sampler is more computationally demanding than the likelihood in the Particle Filters (this is because of the need to compute the L-kernel). Therefore, resampling accounts for a lower percentage of the entire workload in the SMC Sampler than it does in the Particle Filter. The resampling step is no longer such a significant bottleneck for a low number of cores. This is discussed in more detail in the next section.

5.3 Maximum speed-up

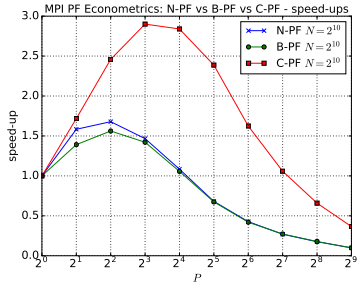
All the previous experiments use relatively simple proposal distributions and likelihoods. However, in real problems, these two tasks are likely to be much more complicated (e.g. they may involve non-linear systems or Partial Differential Equations etc.). In the next experiment, we investigate the impact that a more computationally intensive Importance Sampling step has on the maximum speed-up.
In order to simulate this scenario, we adjust the experiment described in Section 5.2.2 by using $D > 1$ sensors spread over the Cartesian plane. This practice is also common in real applications to make the estimate more accurate (since the triangulation observability criterion is satisfied [23]). The measurement is now a D-dimensional measurement vector:

$$\mathbf{Y}_t = \arctan\left(\frac{X_t^2 - \tilde{y_k}}{X_t^0 - \tilde{x_k}}\right) + \mathbf{W}_t \;\; \forall k = 1, ..., D \tag{17}$$

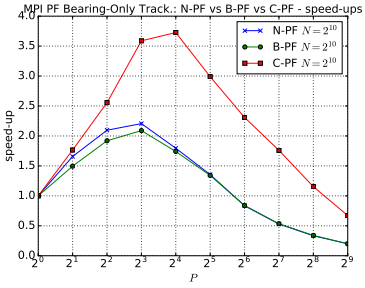
where $(\tilde{x_k}, \tilde{y_k})$ is the position of the k-th sensor with respect to the target. The state equation remains the same as is described in Section 5.2.2 such that M is unchanged. We consider $N = 2^{24}$. The maximum speed-up efficiency for each D is estimated as the speed-up for $P = 512$ vs the ideal speed-up for the same P. We increase D until we have at least 50% efficiency. For each D we also report the percentage of the total workload that Importance Sampling accounts for when the run-time of redistribute is at its peak, i.e. for $P = 2$.
As we can observe in Figure 8, a more computationally intensive Importance Sampling step leads to higher speed-ups. The speed-up for $D = 360$ is indeed about 7.3 times the speed-up for $D = 1$ (which corresponds to the experiment in Section 5.2.2). Therefore, in these problems, the bottleneck for a low number of cores is likely to be the Importance Sampling step and not resampling. However, when $P = N$, Importance Sampling has $O(1)$ time complexity while resampling has complexity of $O((log_2 N)^2)$. In other words, since resampling always emerges as the bottleneck for a sufficiently high level of parallelism, it is crucial to use a parallelisable redistribute such that we can achieve near-linear speed-ups for higher P.

5.4 Space Complexity

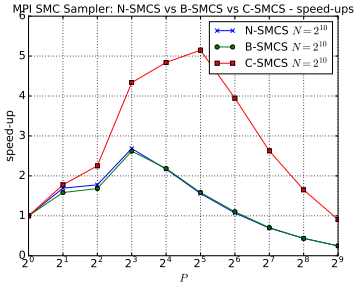
N-R and B-R, have both scalable space complexity equal to $O(M \cdot \frac{N}{P})$. However, C-R has constant space complexity equal to $O(M \cdot N)$ [10]: one core is in charge of collecting the particles, performing the routine locally and then distributing the new population back to the other cores.
The main side effect is that when the available memory in each node is insufficient to store all the necessary data for $P = 1$, we cannot run C-R for any P. In contrast, even for very large values of N, we can always run N-R or B-R as long as each node has enough memory for its data. In order to show this problem, we repeat the experiment described in Section 5.2.2 on Chadwick (which has a 64 GB memory in each node, i.e. less than Barkla's 384 GB per node). Figure 9 shows the measured run-times for $N = 2^{31}$ (speed-ups are not provided since the baseline is impossible to run due to space complexity limitations). The results for $N \in \{2^{10}, 2^{17}, 2^{24}\}$ are left out for brevity since they resemble the results in Figure 7. The total absence of a curve for C-PF or the missing points for N-PF occur because of an mpirun abort (which happens when we request more memory than the node has). As we can see we need at least $P = 64$ cores to run N-PF while it is never possible with C-PF with $N = 2^{31}$.
We can conclude that N-PF/N-SMCS outperforms B-PF/B-SMCS for any P and outperforms C-PF/C-SMCS for $P \geq 32$ cores. Furthermore, it is always possible to run N-PF/N-SMCS while C-PF/C-SMCS may be impossible to run for high N.



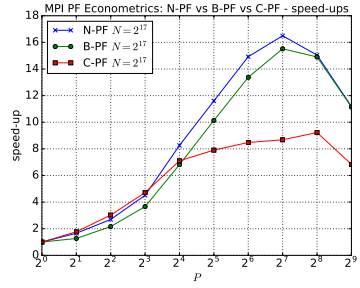
(a) Econometrics for $N = 2^{10}$



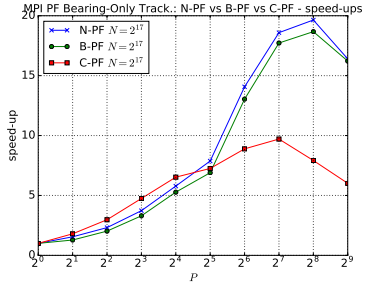
(b) Bearing-Only Track. for $N = 2^{10}$



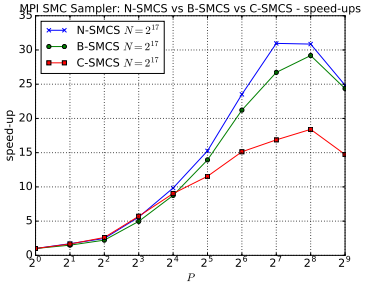
(c) Synthetic SMC Sampler for $N = 2^{10}$



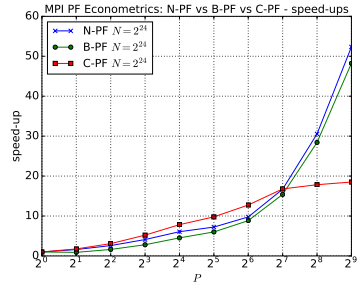
(d) Econometrics for $N = 2^{17}$



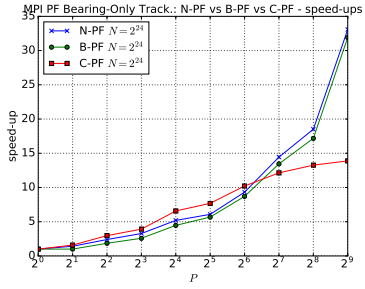
(e) Bearing-Only Track. for $N = 2^{17}$



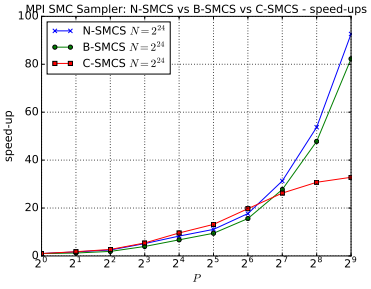
(f) Synthetic SMC Sampler for $N = 2^{17}$



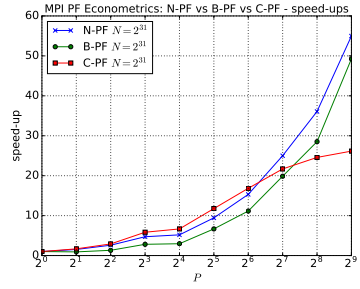
(g) Econometrics for $N = 2^{24}$



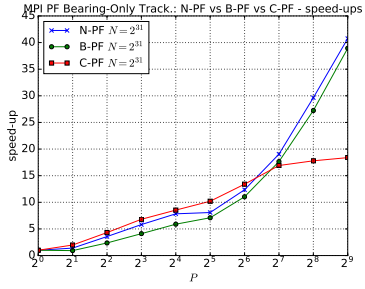
(h) Bearing-Only Track. for $N = 2^{24}$



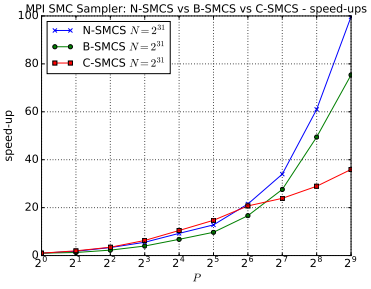
(i) Synthetic SMC Sampler for $N = 2^{24}$



(j) Econometrics for $N = 2^{31}$



(k) Bearing-Only Track. for $N = 2^{31}$



(l) Synthetic SMC Sampler for $N = 2^{31}$

Figure 7: SMC methods: speed-ups vs P for $N = 2^{10}, 2^{17}, 2^{24}, 2^{31}$

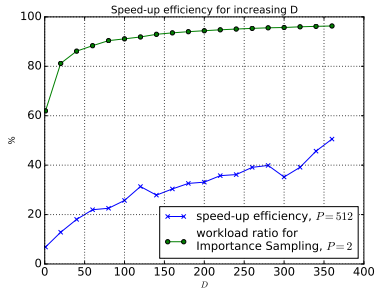


Figure 8: Multi Sensors: max speed-up efficiency for increasing D

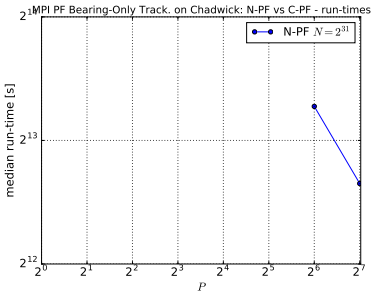


Figure 9: Bearing-Only Track. for $N = 2^{31}$ on Chadwick

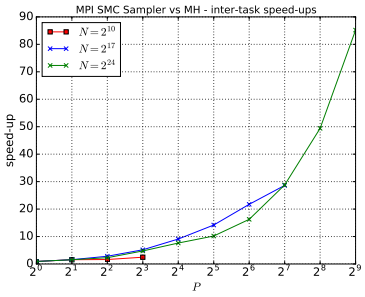


Figure 10: N-SMCS vs MH: inter-task speed-ups vs P for $T_{SMC} = 100$

5.5 SMC Sampler vs Metropolis-Hastings

5.5.1 Description

As we have anticipated in Section 4.4, this experiment aims to achieve two goals. We first want to prove that a P-core implementation of the SMC Sampler can achieve a lower run-time than a single-chain Metropolis-Hastings when both algorithms draw the same number of samples in total (see below). Then we want to prove that the extra speed-up that P cores provide can make an SMC Sampler more accurate than Metropolis-Hastings, since as P increases an SMC Sampler can perform more iterations over the same time span.

The first part of the experiment is done by comparing the run-time of both algorithms for the same workload such that:

$$T_{MH} = N \times T_{SMC} \tag{18}$$

To investigate the second issue, we primarily need to know the inter-task speed-up SU_P which P cores can provide, keeping N fixed. We estimate SU_P from the first part of the experiment. Then we compare the algorithms in terms of accuracy over the same computational time which happens when:

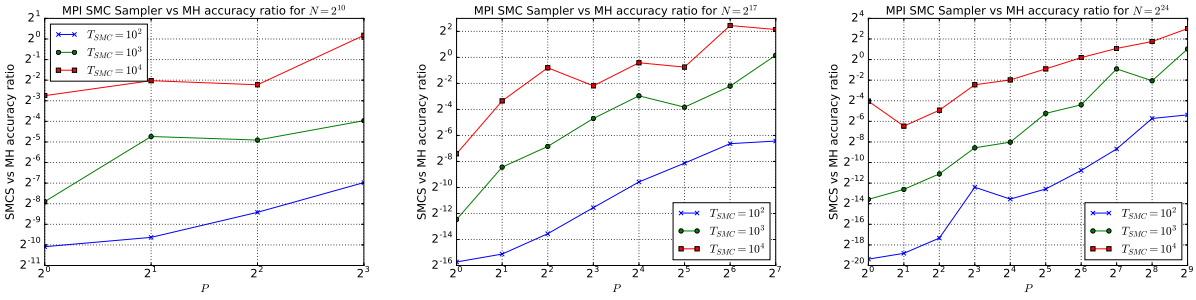
$$T_{MH} = N \times T_{SMC} \times SU_P, \ P = 1, 2, 4, ... \tag{19}$$

In other words, the SMC Sampler will run for SU_P-times more iterations (or less if $0 \leq SU_P \leq 1$) over the same run-time.

The number of samples, N, and the number of cores, P, are the same as in the previous experiments and we will use $T_{SMC} \in \{100, 1000, 10000\}$. T_{MH} is constant, independent of P and always picked using (19) for $P = 1$. The SMC Sampler is once again assessed in the worst-case setting when resampling is needed at every iteration.

5.5.2 Results

Figure 10 shows the inter-task speed-up between SMC Sampler and Metropolis-Hastings for the same workload (see (18)), after having set $T_{SMC} = 100$. We calculate that a single-core implementation of the SMC Sampler is slightly slower (typically by 8%) than Metropolis-Hastings. This means that an SMC Sampler running on a cluster of nodes could be much faster than Metropolis-Hastings. As we can see, for high values of N (which as we have seen lead to larger speed-ups) the SMC Sampler can be up to 85 times faster than Metropolis-Hastings.



(a) $N = 2^{10}$ (b) $N = 2^{17}$ (c) $N = 2^{24}$

Figure 11: SMC Sampler vs Metropolis-Hastings accuracy ratios

In the second part of the experiment we compare the accuracy (expressed as Root Mean Squared Error (RMSE)) for high speed-ups when both algorithms run for the same time span (see (19)). To make the run-time equivalent we set SU_P to

the values shown in Figure 10. Figure 11 shows the accuracy ratios between the algorithms vs P for increasing N or T_{SMC}. The baseline (and numerator of the accuracy ratio) is the accuracy of Metropolis-Hastings (see Table 2). As we can see, for low values of T_{SMC}, the SMC Sampler does not outperform Metropolis-Hastings as the gap for $P = 1$ is initially too big. However, while an SMC Sampler is less accurate, the relative benefit of using Metropolis-Hastings reduces when P increases. Combinations of bigger values for N or T_{SMC} lead to comparable gains in accuracy when SU_P is maximised (see pairs: $N = 2^{10}$, $T_{SMC} = 10^4$; $N = 2^{17}$, $T_{SMC} = 10^3$; $N = 2^{24}$, $T_{SMC} = 10^3$). In the end, for even higher values of T_{SMC} or N the initial gap at the baseline is lower such that, when SU_P increases sufficiently, the SMC Sampler finally outperforms Metropolis-Hastings. Figures 11c proves that the RMSE of the SMC Sampler can be up to approximately 8 times lower. If we consider that the standard deviation σ for Metropolis-Hastings scales as $O(1/\sqrt{N})$, we can infer that the ideal improvement in accuracy for $P = 512$ cores would be:

$$\frac{\sigma_{P=512}}{\sigma_{P=1}} = \frac{\sqrt{N}}{\sqrt{PN}} = \frac{1}{\sqrt{512}} \approx \frac{1}{22.63} \tag{20}$$

This would occur only if we could trivially parallelise a single-chain Metropolis-Hastings and observe linear speed-up, which means that the proposed MPI SMC Sampler already achieves approximately 35% efficiency with respect to the ideal scenario. This is an encouraging finding, especially considering that we have used a simple SMC Sampler. We anticipate that further improvements in accuracy would result from using a more sophisticated L-Kernel, better recycling, novel proposal distributions or alternative resampling implementations [24]. The results for $N = 2^{31}$ are not shown. This is because for high values of T_{SMC}, the run-times of Metropolis-Hastings and the SMC Sampler for low values of P exceed the simulation time limit on both clusters (which is set to 3 days).

Table 1: Details of the clusters.

Name	**Barkla**	**Chadwick**
OS	CentOS Linux 7	RHEL 6.10 (Santiago)
Number of Nodes	16	8
Cores per node	40	16
CPU	2 Xeon Gold 6138	2 Xeon(R) E5-2660
RAM	384 GB	64 GB
MPI Version	OpenMPI-1.10.1	OpenMPI-1.5.3
Max time per job	72 hours	72 hours

Table 2: Metropolis-Hastings: RMSE (log scale)

		T_{SMC}		
		10^2	10^3	10^4
N	2^{10}	-10.12	-11.62	-12.36
	2^{17}	-12.41	-13.06	-13.62
	2^{24}	-13.64	-13.87	-13.95

6 Conclusions

In this paper, we have shown that a parallel implementation of the SMC Sampler on distributed memory architectures is an advantageous alternative to Metropolis-Hastings as it can be up to 85 times faster over the same workload, and up to 8 times more accurate over the same run-time for 512 cores.

To get to this position, we have made several advances. An MPI implementation of the SMC Sampler has previously been unavailable but we have proven that it can be produced by porting the key components of the Particle Filter. There exist several alternative algorithms to perform the common bottleneck, redistribute, including a state-of-the-art parallel algorithm and a textbook non-parallelisable implementation. In this paper, we have optimised the parallel algorithm and proven it can outperform the current approach for any number of cores and be up to 3 times faster than the textbook implementation for a sufficiently high degree of parallelism. In addition, we have demonstrated the infeasibility of the non-parallelisable algorithm for large numbers of particles.

The proposed algorithm for 512 cores is 100 times as fast as its serial configuration in the worst case scenario, which occurs when resampling (and redistribute) is needed at every step and, most importantly, when the model is unrealistically simple and hence Importance Sampling has a very fast constant time. More realistic models have highly computationally intensive proposal distributions or likelihoods. Under these realistic conditions, we have shown that the overall speed-up increases with the workload of Importance Sampling and the maximum recorded speed-up is about 254 for 512 cores.

A key observation we can make is that the SMC Sampler version we have used is a basic reference version as the L-kernel is equal to the proposal distribution, which is Gaussian; better recycling and resampling are yet to be explored. This means that we still have left significant scope for future improvements. A combination of intelligent recycling, a more sophisticated L-Kernel, improved proposal distribution and better resampling may have major impacts on the accuracy.
Another improvement avenue is to speed up the run-time which as we have seen can indirectly improve the accuracy too. One possible way to achieve this goal is to investigate the benefits of mixing shared memory architectures and distributed memory architectures. OpenMP is the most common programming model for shared memory architectures and including OpenMP algorithms within MPI is a routine approach in the high performance computing domain. Data locality may also lead to alternative and more efficient ways of implementing redistribute. A second environment that may lead to further speed-up consists of using the additional computational power that the GPU card within each machine provides.
Future work will focus on implementing all these improvements and comparing the resulting SMC Sampler with better MCMC methods than Metropolis-Hastings. These comparisons must necessarily be made both in the single and multiple chain contexts.

Acknowledgments

This work was supported by the UK EPSRC Doctoral Training Award and by Schlumberger.

Appendices

The following algorithms summarise the routines which are described in detail in Sections 3 and 4 and compared in Section 5 of the main paper. In these algorithms, we use the same notation of the main paper: arrays and matrices are in bold while scalars, such as some input parameters and single elements of one-dimensional arrays, are written in italic font-style.

SMC methods and Metropolis-Hastings

Algorithms 1, 2 and 3 explain the two considered SMC methods and Metropolis-Hastings.

Algorithm 1 SIR Particle Filter

Input: T, N, N^*
Output: $\mathbf{f}_t$

$\mathbf{x}_0, \mathbf{w}_0 \leftarrow$ `Initialisation`(), each particle is initially drawn from the prior distribution $q(\mathbf{x}_0) = p(\mathbf{x}_0)$ and each weight is initialised to $1/N$
for $t \leftarrow 1; t \leq T; t \leftarrow t + 1$ **do**
    $\mathbf{Y}_t \leftarrow$ `New_Measurement`()
    $\mathbf{x}_t, \mathbf{w}_t \leftarrow$ `Importance_Sampling`(), see (3) and (4) in the main paper
    $\tilde{\mathbf{w}}_t \leftarrow$ `Normalise`$(\mathbf{w}_t)$, see (2) in the main paper
    $N_{eff} \leftarrow$ `ESS`$(\tilde{\mathbf{w}}_t)$, see (1) in the main paper
    if $N_{eff} < N^*$ **then**
        $\mathbf{x}_t, \mathbf{w}_t \leftarrow$ `Resampling`$(\mathbf{x}_t, \tilde{\mathbf{w}}_t, N)$
    end if
    $\mathbf{f}_t \leftarrow$ `Mean`$(\mathbf{x}_t, \mathbf{w}_t)$, calculate the weighted mean of the particles to estimate the real state.
end for

Resampling and Redistribute

Algorithm 4 depicts the chosen resampling step for our implementation of the Particle Filter and the SMC Sampler. The three considered MPI implementations of the constituent redistribute step are described by Algorithms 6, 7 and 9. These routines make use of Algorithm 5 to redistribute within each core once the workload is balanced (or centralised as in Algorithm 6). Algorithm 8 explains the single-core Nearly Sort that is used for the MPI Nearly Sort in Algorithm 9 which replaces the MPI Bitonic Sort in Algorithm 7.

Algorithm 2 SMC sampler with recycling

Input: T, N, N^*

Output: $\mathbf{x}_t$, $\hat{\mathbf{f}}$

$\mathbf{x}_0, \mathbf{w}_0 \leftarrow$ `Initialisation`(), $\mathbf{x}_0 \sim q(\mathbf{x}_0)$ and each particle is assigned to its initial weight $w_0^i = \frac{\pi_0(\mathbf{x}_0^i)}{q_0(\mathbf{x}_0^i)}$
for $t \leftarrow 1; t \leq T; t \leftarrow t+1$ **do**
    $\tilde{c}_t \leftarrow$ `Normalisation_Constant`$(\mathbf{w}_t)$, see (8) in the main paper
    $\mathbf{x}_t, \mathbf{w}_t \leftarrow$ `Importance_Sampling`(), see (5) in the main paper for $\mathbf{w}_t$; $\mathbf{x}_t \sim q(\mathbf{x}_t|\mathbf{x}_{t-1})$
    $\tilde{\mathbf{w}}_t \leftarrow$ `Normalise`$(\mathbf{w}_t)$, see (2) in the main paper
    $\mathbf{f}_t \leftarrow$ `Estimate`$(\mathbf{x}_t, \mathbf{w}_t)$, see (7) in the main paper
    $N_{eff} \leftarrow$ `ESS`$(\tilde{\mathbf{w}}_t)$, see (1) in the main paper
    if $N_{eff} < N^*$ **then**
        $\mathbf{x}_t, \mathbf{w}_t \leftarrow$ `Resampling`$(\mathbf{x}_t, \tilde{\mathbf{w}}_t, N)$
    end if
end for
$\hat{\mathbf{f}} \leftarrow$`Recycling`$(\mathbf{f}, \tilde{\mathbf{c}}, T)$, see (6) in the main paper

Algorithm 3 Metropolis-Hastings

Input: T, ϵ, $\mathbf{\Sigma}$

Output: $\mathbf{x}_t$

$\mathbf{x}_0 \sim q(\mathbf{x}_0)$
for $t \leftarrow 1; t \leq T; t \leftarrow t+1$ **do**
    $\mathbf{x}^* \sim N(\mathbf{x}^*|\mathbf{x}_{t-1}, \epsilon^2\mathbf{\Sigma})$, a new sample is drawn from the proposal distribution
    $a = min\{1, \frac{\pi(\mathbf{x}^*)q(\mathbf{x}_{t-1}|\mathbf{x}^*)}{\pi(\mathbf{x}_{t-1})q(\mathbf{x}^*|\mathbf{x}_{t-1})}\}$, calculate the acceptance probability
    $r \sim [0, 1]$
    if $a < r$ **then**
        $\mathbf{x}_t = \mathbf{x}^*$, the proposed sample is accepted
    else
        $\mathbf{x}_t = \mathbf{x}_{t-1}$, the proposed sample is rejected and the old sample is propagated to the next iteration
    end if
end for

References

[1] Pierre Del Moral, Arnaud Doucet, and Ajay Jasra. Sequential monte carlo samplers. *Journal of the Royal Statistical Society. Series B (Statistical Methodology)*, 68(3):411–436, 2006.

[2] Wei Shao, Guangbao Guo, Fanyu Meng, and Shuqin Jia. An efficient proposal distribution for metropolis-hastings using a b-splines technique. *Computational Statistics & Data Analysis*, 57(1):465 – 478, 2013.

[3] 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(5):1305–1319, March 2016.

[4] A. E. Brockwell. Parallel markov chain monte carlo simulation by pre-fetching. *Journal of Computational and Graphical Statistics*, 15(1):246–261, 2006.

[5] Radu V. Craiu, Jeffrey Rosenthal, and Chao Yang. Learn from thy neighbor: Parallel-chain and regional adaptive mcmc. *Journal of the American Statistical Association*, 104(488):1454–1466, 2009.

[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(2):174–188, 2002.

Algorithm 4 Minimum Variance Resampling

Input: $\mathbf{x}_t, \mathbf{w}_t, N$

Output: $\mathbf{x}_t, \mathbf{w}_t$

$\mathbf{ncopies} \leftarrow$ `MVR`$(\mathbf{w}_t)$, apply Minimum Variance Resampling to generate $\mathbf{ncopies}$ from $\mathbf{w}_t$
$\mathbf{x}_t \leftarrow$ `Redistribute`$(N, \mathbf{ncopies}, \mathbf{x}_t)$
$\mathbf{w}_t \leftarrow$ `Reset`$(\mathbf{w}_t)$, all weights are reset to $1/N$

Algorithm 5 Sequential Redistribute (S-R)

Input: N, $\mathbf{ncopies}$, $\mathbf{x}$
Output: $\mathbf{x}_{new}$

$i \leftarrow 0$
for $j \leftarrow 0; j < N; j \leftarrow j + 1$ **do**
    for $k \leftarrow 0; k < ncopies^j; k \leftarrow k + 1$ **do**
        $\mathbf{x}^i_{new} \leftarrow \mathbf{x}^i$
        $i \leftarrow i + 1$
    end for
end for

Algorithm 6 Centralised Redistribute (C-R)

Input: N, $\mathbf{ncopies}$, $\mathbf{x}$, mpi_rank, P
Output: $\mathbf{x}$

if $mpi\_rank == 0$ **then**
    Allocate memory for N integers and N particles
end if
The master core (i.e., the core with $mpi\_rank = 0$) collects the N/P elements in $\mathbf{ncopies}$ from each core using MPI_Gather and stores them into $\mathbf{tmp\_ncopies}$
The master core collects the N/P particles in $\mathbf{x}$ from each core using MPI_Gather and stores them into $\mathbf{tmp\_x}$
if $mpi\_rank == 0$ **then**
    $\mathbf{tmp\_x}_{new} \leftarrow$ `Sequential_Redistribute`$(N, \mathbf{tmp\_ncopies}, \mathbf{tmp\_x})$, see Algorithm 5
end if
The master core scatters $\mathbf{tmp\_x}_{new}$ to the other cores using MPI_Scatter. $\mathbf{x}$ is used as received buffer
return $\mathbf{x}$

[7] N.J. Gordon, D.J. Salmond, and A.F.M. Smith. Novel approach to nonlinear/non-gaussian bayesian state estimation. *Radar and Signal Processing, IEE Proceedings F*, 140:107 – 113, 05 1993.

[8] S. Maskell, B. Alun-Jones, and M. Macleod. A single instruction multiple data particle filter. In *IEEE Nonlinear Statistical Signal Processing Workshop*, pages 51–54, 2006.

[9] Jeyarajan Thiyagalingam, Lykourgos Kekempanos, and Simon Maskell. Mapreduce particle filtering with exact resampling and deterministic runtime. *EURASIP Journal on Advances in Signal Processing*, 2017(1):71, Oct 2017.

[10] A. Varsi, L. Kekempanos, J. Thiyagalingam, and S. Maskell. Parallelising particle filters with deterministic runtime on distributed memory systems. *IET Conference Proceedings*, pages 11–18, 2017.

Algorithm 7 Bitonic Sort Based Redistribute (B-R)

Input: $\mathbf{Node} = [\mathbf{ncopies}, \mathbf{x}]$, N, P, $n = \frac{N}{P}$
Output: $\mathbf{x}$

if $P > 1$ **then**
    `MPI_Bitonic_Sort`$(\mathbf{Node}, N, P)$
end if
procedure DISTRIBUTE($\mathbf{Node}, N, P, n$)
    if $N == n$ **then**, the workload is now fully balanced as each core has $n = \frac{N}{P}$ particles to copy
        $\mathbf{x} \leftarrow$ `Sequential_Redistribute`$(n, \mathbf{ncopies}, \mathbf{x})$, see Algorithm 5
        return $\mathbf{x}$
    end if
    $\mathbf{csum} \leftarrow$ `MPI_Cumulative_Sum`$(N, P, \mathbf{ncopies})$, MPI_Scan is used between the MPI nodes
    $pivot \leftarrow$ `Pivot_Calc`$(\mathbf{ncopies}, \mathbf{csum})$, $pivot$ is the first index of $\mathbf{csum}$ such that $csum^{pivot} \geq N/2$
    $r \leftarrow pivot - \left(\frac{N}{2} - 1\right)$
    $(\mathbf{Leaf_l}, \mathbf{Leaf_r}) \leftarrow$ `MPI_Rotational_Shifts`$(\mathbf{Node}, r)$, up to $log_2 P$ MPI_Sendrecv since r is expressed as a sum of power of 2 numbers.
    `Distribute`$(\mathbf{Leaf_l}, N/2, P/2, n)$, the left node becomes a new father node and the size of the problem is halved
    `Distribute`$(\mathbf{Leaf_r}, N/2, P/2, n)$, the right node becomes a new father node and the size of the problem is halved
end procedure

Algorithm 8 Sequential Nearly Sort (S-NS)

Input: N, $\mathbf{ncopies}$, $\mathbf{x}$
Output: $\mathbf{x}_{new}, \mathbf{ncopies}_{new}$

$l \leftarrow 0, r \leftarrow N-1$
for $i \leftarrow 0; i < N; i \leftarrow i+1$ **do**
  if $ncopies^i > 0$ **then**
    $ncopies^r_{new} \leftarrow ncopies^i$
    $\mathbf{x}^r_{new} \leftarrow \mathbf{x}^i$
    $r \leftarrow r-1$
  else
    $ncopies^l_{new} \leftarrow ncopies^i$
    $\mathbf{x}^l_{new} \leftarrow \mathbf{x}^i$
    $l \leftarrow l+1$
  end if
end for

Algorithm 9 Nearly Sort Based Redistribute (N-R)

Input: $\mathbf{Node} = [\mathbf{ncopies}, \mathbf{x}]$, N, P, $n = \frac{N}{P}$
Output: $\mathbf{x}$

if $P > 1$ **then**
  `MPI_Nearly_Sort`$(\mathbf{Node}, N, P)$
end if

procedure DISTRIBUTE($\mathbf{Node}, N, P, n$)
  if $N == n$ **then**, the workload is now fully balanced as each core has $n = \frac{N}{P}$ particles to copy
    $\mathbf{x} \leftarrow$ `Sequential_Redistribute`$(n, \mathbf{ncopies}, \mathbf{x})$, see Algorithm 5
    return $\mathbf{x}$
  end if
  $\mathbf{csum} \leftarrow$ `MPI_Cumulative_Sum`$(N, P, \mathbf{ncopies})$, MPI_Scan is used between the MPI nodes
  $pivot \leftarrow$ `Pivot_Calc`$(\mathbf{ncopies}, \mathbf{csum})$, $pivot$ is the first index of $\mathbf{csum}$ such that $csum^{pivot} \geq N/2$
  $r \leftarrow pivot - \left(\frac{N}{2} - 1\right)$
  $(\mathbf{Leaf_l}, \mathbf{Leaf_r}) \leftarrow$ `MPI_Rotational_Shifts`$(\mathbf{Node}, r)$, up to $log_2 P$ MPI_Sendrecv since r is expressed as a sum of power of 2 numbers.
  `Distribute`$(\mathbf{Leaf_l}, N/2, P/2, n)$, the left node becomes a new father node and the size of the problem is halved
  `Distribute`$(\mathbf{Leaf_r}, N/2, P/2, n)$, the right node becomes a new father node and the size of the problem is halved
end procedure

[11] Metropolis NS, A.W. Rosenbluth, M.N. Rosenbluth, AH Teller, and E J. Teller. Equation of state calculations by fast computing machines. 21:1087–1092, 01 1953.

[12] Jianye Ching and Yi-Chu Chen. Transitional markov chain monte carlo method for bayesian model updating, model class selection, and model averaging. 133, 07 2007.

[13] Simon Duane, A.D. Kennedy, Brian J. Pendleton, and Duncan Roweth. Hybrid monte carlo. *Physics Letters B*, 195(2):216 – 222, 1987.

[14] J. D. Hol, T. B. Schon, and F. Gustafsson. On resampling algorithms for particle filters. In *2006 IEEE Nonlinear Statistical Signal Processing Workshop*, pages 79–82, Sept 2006.

[15] 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. In *9th IET Data Fusion Target Tracking Conference*, pages 1–8, May 2012.

[16] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, 1980.

[17] Kenneth E. Batcher. Sorting networks and their applications. In *Proceedings of the 1968 Spring Joint Computern Conference (SJCC)*, volume 32, pages 307–314. AFIPS Press, 1968.

[18] M. S. Paterson. Improved sorting networks with o(logn) depth. *Algorithmica*, 5(1):75–92, Jun 1990.

[19] J. Seiferas. Sorting networks of logarithmic depth, further simplified. *Algorithmica*, 53(3):374–384, Mar 2009.

[20] M. Ajtai, J. Komlós, and E. Szemerédi. An 0(n log n) sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 1–9, New York, NY, USA, 1983. ACM.

[21] S.W.A.H. Baddar and K.E. Batcher. *Designing Sorting Networks: A New Paradigm*. SpringerLink : Bücher. Springer New York, 2012.

[22] Arnaud Doucet, Mark Briers, and Stéphane Sénécal. Efficient block sampling strategies for sequential monte carlo methods. *Journal of Computational and Graphical Statistics*, 15(3):693–711, 2006.

[23] Darko Mušicki. Bearings only multi-sensor maneuvering target tracking. *Systems & Control Letters*, 57(3):216 – 221, 2008.

[24] T. Li, M. Bolic, and P. M. Djuric. Resampling methods for particle filtering: Classification, implementation, and strategies. *IEEE Signal Processing Magazine*, 32(3):70–86, 2015.