# A Shared Memory Optimal Parallel Redistribution Algorithm for SMC Samplers with Variable Size Samples

Alessandro Varsi[1][0000−0003−2218−4720], Efthyvoulos Drousiotis[1][0000−0002−9746−456X], Paul G. Spirakis[2][0000−0001−5396−3749], and Simon Maskell[1][0000−0003−1917−2913]

[1] University of Liverpool, Department of Electrical Engineering and Electronics, Liverpool, UK, L69 3GJ {A.Varsi, E.Drousiotis, SMaskell}@liverpool.ac.uk
[2] University of Liverpool, Department of Computer Science, Liverpool, UK, L69 3GJ P.Spirakis@liverpool.ac.uk

**Abstract.** Sequential Monte Carlo (SMC) samplers are Bayesian inference methods to draw $N$ random samples from challenging posterior distributions. Their simplicity and competitive accuracy make them popular in various applications of Machine Learning (ML), Bayesian Optimization (BO), and Statistics. In many applications, run-time is critical under strict accuracy requirements, making parallel computing essential. However, an efficient parallelization of SMC depends on how effectively its bottleneck, the redistribution step, is parallelized. This is hard due to workload imbalance across the cores, especially when the samples are of variable-size. A parallel redistribution for variable-size samples was recently proposed for Shared Memory (SM) architectures. This method resizes all samples to the size of the biggest sample, $\overline{M}$, and constrains the samples to be indivisible, i.e., forces the cores to redistribute whole samples. This leads to inefficient run-time, and a sub-optimal time complexity, $O(\overline{M} \log_2 N)$. This study addresses the challenge of Optimal Parallel Redistribution (OPR) for variable-size samples. We first prove that OPR for indivisible variable-size samples is NP-complete. Then, we present an OPR algorithm for SM that does not resize the samples and allows cores to redistribute either whole samples or fractions of them. We prove theoretically that this approach achieves optimal $O(\hat{M} \log_2 N)$ time complexity, where $\hat{M}$ is the average size of the redistributed samples. We also show experimentally that the proposed approach is up to $10\times$ faster than the reference method on a 32-core SM machine.

**Keywords:** Parallel Algorithms · Sequential Monte Carlo · Parallel Redistribution · Shared Memory · Parallel Methods for ML and BO

## 1 Introduction

### 1.1 Motivation

Sequential Monte Carlo (SMC) samplers are commonly used Bayesian inference methods to make estimations of the state of a statistical model, $\mathbf{X}$, given some

data, $\mathbf{Y}$. The key idea is to generate randomly $N$ weighted, and statistically independent samples from the posterior distribution of the model, $\pi(\mathbf{X}, \mathbf{Y})$, and use the samples to make estimates. The simplicity and the state-of-art performance of SMC samplers make them a popular choice in many application domains, e.g. Machine Learning (ML) [1], Bayesian Optimization (BO) [2,3], and Medicine [4].

Several ideas have been explored to make the estimations of SMC samplers more accurate. Examples include using better proposal distributions [5–7], tempering [8], or simply increasing $N$ [9,10]. However, while these ideas are all valid and generally applicable, they share a common side effect: the SMC sampler becomes more computationally intensive. This side effect is more problematic in time-crucial applications, such as Epidemiology [11] (e.g., when computing the R-number of viruses, such as for COVID-19) or Crime Prediction [12]. Hence, parallel computing becomes crucial to trade-off accuracy and run-time.

### 1.2 Problem Definition and Related Work

Since the samples are statistically independent, they can be generated randomly in embarrassingly parallel fashion, which makes SMC an appealing alternative to other Bayesian inference methods, such as Markov Chain Monte Carlo [13]. However, at some point, the samples may experience a numerical error, called degeneracy, which makes the samples (and the state estimates) diverge from the true state of interest. This error is corrected by using a resampling algorithm, which replaces the samples that have diverged by redistributing copies of the samples that have not. Resampling is widely known to be the parallelization bottleneck of SMC samplers. This is due to the issues encountered when parallelizing the redistribution sub-task of resampling, in such a way that the workload is optimally distributed across $P$ processing elements (or cores, the terms are used interchangeably here). For clarity, we provide the following definition:

**Definition 1 (Optimal Parallel Redistribution (OPR)).** *The problem of redistributing $N$ samples with $P$ parallel cores is solved optimally if and only if the maximum workload on any core is minimized, where the workload of a core is defined as the sum of the copying costs for the samples assigned to that core.*

In the special case where all samples have the same size, $M$, parallel redistribution has been extensively studied and solved optimally, achieving lower bound time complexity, $O(\log_2 N)$. Examples can be found for both Distributed Memory (DM) [14,15] and Shared Memory (SM) architectures [9,16,17].

In the most general case, the samples may have variable sizes. This occurs in several areas of ML and BO, where the model is designed to sample and estimate abstract data types, such as Decision Trees (DTs) [18–20], Additive Structures [21], and Time Series Structures [22]. In this scenario, OPR is significantly more complicated to achieve than in the fixed-size scenario, as the workload is even more unbalanced. Indeed, variable-size samples introduce an additional layer of complexity: balancing workloads now depends not only on the number of samples assigned to each core but also on the cost of copying each individual sample, which can vary significantly. The work in [23] describes a parallel SMC sampler for DTs on SM, in which the redistribution step is centralized

to a single core, achieving little to no speed-up. In [24], a parallel redistribution for variable-size samples is proposed for SM and then ported to DM [25]. This approach first resizes all samples until they match the size of the biggest sample, $\overline{M}$, and then performs parallel redistribution using conventional parallel techniques for redistributing $N$ fixed-size samples, where the cores redistribute an equal (integer) number of samples. The approach in [24] achieves $O(\overline{M}\log_2 N)$ time complexity, and is, therefore, sub-optimal since the cost of computation per sample is bound to the worst case, $O(\overline{M})$. This consequently poses the following question: is OPR achievable when redistributing $N$ variable-size samples?

### 1.3 Paper Contribution and Outline

In this paper, we first prove that if all cores are constrained to redistribute an integer number of variable-size samples, OPR is NP-complete. In other words, under this constraint, OPR for variable-size samples is achievable in polynomial time only if P=NP. This is the case for DM systems and SM approaches like [24]. Then we propose a novel SM parallel redistribution algorithm for variable-size samples in which the samples are divisible (i.e., the cores may either redistribute whole samples or fractions of them) and do not need to be resized. Our theoretical analysis proves that our approach achieves optimal $O(\hat{M}\log_2 N)$ parallel time complexity, where $\hat{M}$ is the average size of the samples to be redistributed.

We also present experimental results comparing two versions of the same SMC sampler, differing only in their parallel redistribution approach. The results demonstrate that our approach achieves a significantly faster run-time than the approach described in [24], by roughly a $10\times$ factor.

The rest of the paper is organized as follows: in Section 2, we first describe SMC samplers in general, and then we briefly explain how SMC is parallelized in [24]. Section 3 proves that OPR is NP-complete if the cores are constrained to redistribute an integer number of variable-size samples. In Section 4, we present and analyze our approach in detail. Section 5 shows the numerical results. Section 6 concludes the paper and proposes possible future research directions.

## 2 Background and Previous Work

In this section, we first introduce the SMC sampler. For details, the reader is referred to [26]. Then we provide brief details on how SMC is parallelized in [24].

### 2.1 Sequential Monte Carlo Samplers

The SMC sampler generates $N$ random samples of $\pi(\mathbf{X},\mathbf{Y}) = p(\mathbf{Y}|\mathbf{X})p(\mathbf{X})$, i.e., the posterior distribution of the model, and $p(\mathbf{X})$ and $p(\mathbf{Y}|\mathbf{X})$ are the prior and the likelihood, respectively. SMC performs $K$ iterations, for $k = 0, 1, \ldots, K-1$, to generate and update $N$ random samples of the posterior, $\mathbf{x}_k^i$, for $i = 0, 1, \ldots, N-1$. However, since $\pi(\cdot,\cdot)$ is often impossible to be sampled from directly, the SMC sampler uses a convenient and arbitrary proposal distribution (or simply

proposal) to generate the samples, and then gives each sample a weight, $\mathbf{w}_k^i$, which quantifies how well $\mathbf{x}_k^i$ approximates the true state, $\mathbf{X}$.

At the first iteration, $k = 0$, each sample, $\mathbf{x}_0^i$, is sampled from the prior, and its weight is initialized to $\mathbf{w}_0^i = \pi(\mathbf{x}_0^i, \mathbf{Y})/p(\mathbf{x}_0^i)$. After that, for all $k > 0$, the SMC sampler draws new samples and updates their weights as follows:

$$\mathbf{x}_k^i \sim q(\cdot|\mathbf{x}_{k-1}^i), \tag{1a}$$

$$\mathbf{w}_k^i = \mathbf{w}_{k-1}^i \frac{\pi\left(\mathbf{x}_k^i, \mathbf{Y}\right)}{\pi\left(\mathbf{x}_{k-1}^i, \mathbf{Y}\right)} \frac{q\left(\mathbf{x}_{k-1}^i|\mathbf{x}_k^i\right)}{q\left(\mathbf{x}_k^i|\mathbf{x}_{k-1}^i\right)}, \tag{1b}$$

where $q(\cdot|\cdot)$ is the proposal. We note that Equation (1) is often referred to as Importance Sampling (IS) in the literature.

After (1), the weights need to be normalized as follows:

$$\tilde{\mathbf{w}}_k^i = \frac{\mathbf{w}_k^i}{\sum_{j=0}^{N-1} \mathbf{w}_k^j}, \tag{2}$$

so that $\sum_i \tilde{\mathbf{w}}_k^i = 1$ and $\tilde{\mathbf{w}}_k^i$ can approximate expectations of the posterior.

In theory, this approach correctly represents the posterior as $N \to \infty$. In practice, IS suffers from degeneracy, a numerical error caused by having the samples generated from the proposal, instead of directly from the posterior. This leads to all weights but one to drop to 0 within a few iterations, making the samples diverge from the posterior. To correct this error, the typical approach is to perform the resampling algorithm if the effective sample size,

$$N_{\text{eff}} = \frac{1}{\sum_{i=0}^{N-1} \left(\tilde{\mathbf{w}}_k^i\right)^2}, \tag{3}$$

drops below an arbitrary threshold, normally set to $0.5N$. Resampling replaces the samples with low weights with copies of the samples with high weights.

Many resampling schemes exist in the literature. Here, we use systematic resampling [27], arguably the most common scheme. This resampling variant regenerates the sample population by performing the following three steps.

*Step 1 - Choice.* This step determines how many times each sample, $\mathbf{x}_k^i$, will be copied. In systematic resampling, this choice involves computing $\mathbf{cdf} \in \mathbb{R}^N$, the cumulative sum (or prefix sum, the terms are used interchangeably) of the normalized weights as follows:

$$\mathbf{cdf}^i = N \sum_{j=0}^{i-1} \tilde{\mathbf{w}}_k^j, \quad \forall i = 0, 1, 2, \ldots, N-1. \tag{4}$$

Each sample, $\mathbf{x}_k^i$, for $i = 0, 1, \ldots, N-1$, will then be copied as many times as

$$\mathbf{ncopies}^i = \lceil \mathbf{cdf}^i + N\tilde{\mathbf{w}}_k^i - u \rceil - \lceil \mathbf{cdf}^i - u \rceil, \tag{5}$$

where $u \sim \texttt{Uniform}[0, 1)$, and $\lceil \cdot \rceil$ is the ceiling operator. From (4) and (5), it can be inferred that $\mathbf{ncopies} \in \mathbb{N}_0^N$, where $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$, and complies with

$$\sum_{i=0}^{N-1} \mathbf{ncopies}^i = N. \tag{6a}$$

$$0 \leq \mathbf{ncopies}^i \leq N, \tag{6b}$$

*Step 2 - Redistribution.* This step is in charge of actually replicating each sample as many times as in (5), such that the samples for which $\mathbf{ncopies}^i = 0$ will be removed. Algorithm 1 describes a textbook implementation of a Sequential Redistribution (S-R). We note that every resampling variant requires redistribution. Therefore, since this paper presents an OPR for variable-size samples, the novelty we propose is generally applicable to any resampling variant.

**Algorithm 1** Sequential Redistribution (S-R)

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

```
i ← 0
for j ← 0; j < N; j ← j + 1 do
    for copy ← 0; copy < ncopies^j; copy ← copy + 1 do
        x_new^i ← x^j
        i ← i + 1
    end for
end for
```

*Step 3 - Reset.* After redistribution, the weights are simply reset to $1/N$, and a new iteration starts over from (1).

After $K$ iterations, the samples are assumed to have converged to the true posterior, and the true state can be estimated as a weighted mean of the samples.

### 2.2 Previous Work on Parallel SMC for Variable Size Samples

Equations (1), (5), and the *Reset* step in resampling are embarrassingly parallel, as these operations are all element-wise tasks. On SM, these tasks can be parallelized by equally dividing the iteration space across $P$ SM threads (i.e., the processing elements), such that each thread performs $O(\frac{N}{P})$ iterations.

Equations (2), (3) both require the computation of a vector sum. This operation is notoriously parallelizable by using reduction, which is well-known to achieve $O(\frac{N}{P} + \log_2 P)$ time complexity.

The prefix sum in Equation (4) can also be performed in $O(\frac{N}{P} + \log_2 P)$ by using prefix reduction [28–30].

Algorithm 1 takes $O(N)$ sequentially, as Equation (6a) holds. However, this algorithm is impossible to be parallelized if using embarrassingly parallel techniques. The main reason is that the workload associated with replicating each sample, $\mathbf{x}^i$, a total of $\mathbf{ncopies}^i$ times is inherently unbalanced as Equation (6b) holds. Furthermore, the workload may be even more unbalanced in the general case in which the samples have variable sizes, which is the scenario we focus on in this paper. A SM parallel redistribution for variable-size samples has been proposed in [24] and is illustrated in Algorithm 2. Brief details follow.

Algorithm 2 makes all cores redistribute the same (integer) number of samples, $\frac{N}{P}$. Therefore, this means that the samples must be assumed to be indivisible, whose definition is formally given as follows.

**Assumption 1 (Indivisibility)** *Each variable-size sample, $\mathbf{x}^i$, must be assigned entirely to a single core, i.e., cannot be split or divided across the memory spaces of multiple cores. In other words, if sample $\mathbf{x}^i$ of size $\mathbf{M}^i$ is assigned to core id, then the entirety of the $\mathbf{M}^i$ sample values contributes to the workload of core id.*

Under these conditions, the workload is likely unbalanced due to the uneven sample sizes. Algorithm 2 then bypasses this problem by resizing all the samples to size, $\overline{M}$, i.e., the size of the biggest sample. This step is parallelized in $O(\overline{M} \log_2 N)$. After that, the samples can be redistributed in $O(\overline{M} \log_2 N)$ by using known parallel redistribution algorithms for fixed-size samples [17]. More precisely, the $P$ threads first compute $\mathbf{csum} \in \mathbf{N}_0^N$, the prefix sum of **ncopies**. Then, each thread can use **csum** to identify a subset of $\frac{N}{P}$ samples to redistribute in $O(\frac{N}{P})$ with a constant time per sample of $O(\overline{M})$ by using Algorithm 1. Ultimately, the original size of the redistributed samples is restored in $O(\overline{M}\frac{N}{P})$ parallel time. Overall, [24] proves Algorithm 2 achieves a sub-optimal $O(\overline{M} \log_2 N)$ time complexity because the computational cost per sample is bounded to the worst case, $O(\overline{M})$. This conclusion leads to the following question: is it possible to achieve OPR on $N$ variable-size, indivisible samples in polynomial time? The next section provides an answer to this question.

**Algorithm 2** Reference Parallel Redistribution for Variable Size Samples

**Input:** $\mathbf{x}$, **ncopies**, $N$, $P$, $n - \frac{N}{P}$
**Output:** $\mathbf{x}$

```
M̄ ←Max(dims(x), P) //dims returns the dimension of x^i ∀i
x ←Pad(x, M̄, P) //Appends up to M̄ − 1 Not a Numbers to each sample
csum ←Prefix Sum(ncopies, P), Begin redistribution
Spawn P threads with id = 0, 1, ..., P − 1{
    p ←Binary Search(csum, ncopies, n, P)
    cp ← min (csum^p − id × n, n)
    x^{n:n×cp−1} ← x^p
    x ← S-R(x^{p+1:N−1}, ncopies^{p+1:N−1}, n − cp)
} //End redistribution
x ← Restore(x, M̄, P)
```

## 3 OPR for Variable Size Indivisible Samples is NP-complete

**Theorem 1.** *Under Assumption 1, the problem of Optimal Parallel Redistribution (OPR), defined as distributing $N$ variable-size samples across $P \leq N$ cores such that the maximum workload among all cores is minimized, is NP-complete.*

*Proof.* To prove Theorem 1, we show OPR under Assumption 1 is both in NP and NP-hard.

*OPR is in NP:* By Assumption 1, each sample is assigned entirely to a single core. Given a proposed solution (redistribution of samples across $P$ cores, for $id = 0, 1, \ldots P-1$), we can compute the workload for each core:

$$\mathbf{W}^{id} = \sum_{i\text{-th samples assigned to } id} \mathbf{ncopies}^i \mathbf{M}^i,$$

where $\mathbf{M}^i$ is the size of the $i$-th sample. The maximum workload is:

$$\overline{W} = \max_{0 \leq id \leq P-1} \mathbf{W}^{id}.$$

Verifying whether $\overline{W}$ is less than or equal to a given threshold can be done in polynomial time. Thus, OPR is in NP.

*OPR is NP-hard:* We reduce the *Partition Problem*, a known NP-complete problem [31], to OPR. The Partition Problem is defined as follows: given a set of positive integers $\mathbf{S} = \{a_0, a_1, \ldots, a_{N-1}\}$, determine if there exists a partition of $\mathbf{S}$ into two subsets, $\mathbf{S}_1$ and $\mathbf{S}_2$, such that:

$$\sum_{x \in \mathbf{S}_1} x = \sum_{y \in \mathbf{S}_2} y = \frac{1}{2} \sum_{z \in \mathbf{S}}.$$

Given an instance of the Partition Problem, we construct an OPR instance with $P = 2$ cores, sample sizes $\mathbf{M}^i = a_i$, for $i = 0, 1, \ldots, N-1$, and $\mathbf{ncopies}^i = 1$ (i.e., each sample is replicated exactly once). The goal in OPR is to minimize:

$$\overline{W} = \max \left( \sum_{x_i \in \mathbf{S}_1} \mathbf{M}^i, \sum_{x_j \in \mathbf{S}_2} \mathbf{M}^j \right).$$

If the Partition Problem has a solution, there exists a distribution of samples across two cores such that $\overline{W} = \frac{1}{2} \sum_{z \in \mathbf{S}}$. Conversely, if OPR achieves $\overline{W} = \frac{1}{2} \sum_{z \in \mathbf{S}}$, the Partition Problem has a solution. Thus, OPR is NP-hard for $P = 2$.

For $P > 2$, OPR generalizes the Partition Problem to the *Multiprocessor Scheduling Problem*, where the objective is to minimize $\overline{W}$ across $P$ cores, subject to Assumption 1. The Multiprocessor Scheduling Problem is known to be NP-hard, and the indivisibility constraint preserves the hardness of OPR.

Since OPR is in NP and NP-hard, we conclude that OPR is NP-complete. □

## 4 A SM OPR Algorithm for Variable Size Samples

Theorem 1 proves that OPR for $N$ variable-size, indivisible samples cannot be solved in polynomial time, unless P=NP. However, while Assumption 1 is indeed mandatory on DM (as the samples are by definition allocated in different memories), it is optional on SM (as the samples are all stored in the same physical memory). In this section, we prove that on SM it is possible to optimally redistribute $N$ variable-size samples in parallel, if Assumption 1 is relaxed.

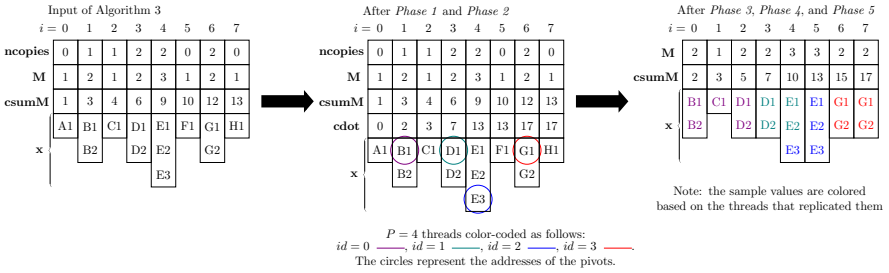



Fig. 1: Algorithm 3 - example for $N = 8$ and $P = 4$. Each sample value is encoded with a letter and a number for brevity.

To achieve this goal, the key idea is to allow the SM threads to redistribute whole samples and/or (potentially) fractions of certain samples. More precisely, if a $M$-dimensional sample, $\mathbf{x}^i$, needs to be copied by a certain thread with identification number, $id$, depending on how the workload is distributed across the $P$ threads, thread $id$ might have to either copy the whole sample, $\mathbf{x}^i$, or copy only $M_s < M$ sample values of $\mathbf{x}^i$, such that thread $id+1$ will have to copy the remaining $M - M_s$ sample values of $\mathbf{x}^i$.

The first thing we need to do is to restructure $\mathbf{x}$ as a one-dimensional (1-D) array of total size $\sum_{i=0}^{N-1} \mathbf{M}^i$, instead of being an array of $N$ samples, each sample being a list of $\mathbf{M}^i$ values, as in Algorithm 2. This means that, once redistribution is complete, $\mathbf{x}$ will have a new total size equal to

$$W_{\text{tot}} = \sum_{i=0}^{N-1} \mathbf{ncopies}^i \mathbf{M}^i, \tag{7}$$

which can also be viewed as the total workload (or total number of memory copies) to redistribute $\mathbf{x}$. Since we work with a 1-D array, we need to be able to address each (variable-size) sample in $O(1)$. We do this by storing in memory $\mathbf{csumM} \in \mathbb{N}_0^N$, i.e., the cumulative sum of $\mathbf{M}$, such that

$$\mathbf{csumM}^i = \sum_{j=0}^{i} \mathbf{M}^j, \quad \forall i = 0, 1, 2, \ldots, N-1, \tag{8}$$

can be considered the base address of $i$-th sample in a 1-D array of variable-size samples. Broadly speaking, if $\mathbf{x}$ is an array of variable-size lists, $\mathbf{x}^{i,m}$ is the $m$-th value of the $i$-th sample, while the same value will be found in index $\mathbf{csumM}^i + m$ if $\mathbf{x}$ is restructured as a 1-D array. With these given inputs, it is possible to prove that redistribution can be parallelized in $O(\frac{N}{P} + \log_2 N)$ with a much lower cost of computation per sample compared to Algorithm 2. Further details follow.

*Phase 1 - Prefix Dot Product.* Equation (7) expresses the total workload to redistribute $\mathbf{x}$ as the dot product of **ncopies** and $\mathbf{M}$. This suggests that we could compute $\mathbf{cdot} \in \mathbb{N}_0^N$, the cumulative dot product of **ncopies** and $\mathbf{M}$, such that each $\mathbf{cdot}^i$, for all $i = 0, 1, \ldots, N-1$, will contain the required workload, as a function of the size of each sample, to redistribute the samples up to the $i$-th index. More precisely, we compute:

$$\mathbf{cdot}^i = \sum_{j=0}^{i-1} \mathbf{ncopies}^j \mathbf{M}^j, \quad \forall i = 0, 1, 2, \ldots, N-1. \tag{9}$$

From (7) and (9), it is relatively straightforward to infer that $W_{\text{tot}} = \mathbf{cdot}^{N-1}$.

*Phase 2 - Workload Split.* We now want to divide the workload as evenly as possible across the $P$ parallel threads, $id = 0, 1, \ldots, P-1$. Since the threads need to redistribute $W_{\text{tot}}$ sample values in total, in theory, every thread could search for a pivot index, $p$, which is the first index such that $\mathbf{cdot}^p \geq id\frac{W_{\text{tot}}}{P}$, and then redistribute $\frac{W_{\text{tot}}}{P}$ sample values starting from its pivot. However, in practice, $W_{\text{tot}}$ might not be an integer factor of $P$, since the samples have variable size. Therefore, we decide to split the workload in such a way each thread will either redistribute $\lfloor\frac{W_{\text{tot}}}{P}\rfloor$ or $\lfloor\frac{W_{\text{tot}}}{P}\rfloor + 1$ sample values, where $\lfloor\cdot\rfloor$ is the floor operator, such that $\lfloor\frac{W_{\text{tot}}}{P}\rfloor$ is the quotient of $W_{\text{tot}} \div P$. More precisely, the workload, $\mathbf{W}^{id} \in \mathbb{N}_0$, to be done by the thread $id$ is computed as follows:

$$\mathbf{W}^{id} = \begin{cases} \lfloor\frac{W_{\text{tot}}}{P}\rfloor + 1, & \text{if } id + 1 \leq W_{\text{tot}} \bmod P, \\ \lfloor\frac{W_{\text{tot}}}{P}\rfloor, & \text{if } id + 1 > W_{\text{tot}} \bmod P, \end{cases} \tag{10}$$

where $\bmod$ is the remainder of $W_{\text{tot}} \div P$. In order to find the pivots, each thread, $id$, must compute, $\mathbf{cw}^{id}$, the cumulative sum of all sample values being redistributed by all threads having a lower identification number than $id$. One could use the exclusive prefix sum of $\mathbf{W}$ to compute $\mathbf{cw}^{id}$. However, the same could be computed in $O(1)$ by using the information in Equation (10) as follows:

$$\mathbf{cw}^{id} = \begin{cases} id \times \mathbf{W}^{id}, & \text{if } id + 1 \leq W_{\text{tot}} \bmod P, \\ \lfloor\frac{W_{\text{tot}}}{P}\rfloor(\mathbf{W}^{id} + 1) + (id - \lfloor\frac{W_{\text{tot}}}{P}\rfloor)\mathbf{W}^{id}, & \text{if } id + 1 > W_{\text{tot}} \bmod P. \end{cases} \tag{11}$$

Each thread, $id$, can then independently search for the first index, $p$, such that

$$\mathbf{cdot}^p \geq \mathbf{cw}^{id}. \tag{12}$$

Binary Search (BS) can be used to search for the pivots since **cdot** is inherently sorted as the output of every prefix reduction is always monotonically increasing.

*Phase 3 - Copying samples.* Each thread $id$ can now independently copy the $\mathbf{W}^{id}$ sample values starting from the pivot index found in the previous step. At any given iteration, a sample value will actually be copied if and only if it happens to be a quantity of any $i$-th sample such that $\mathbf{ncopies}^i > 0$.

*Phase 4 - Redistribution of sample sizes.* After the previous step, the samples have been redistributed. However, we also need to guarantee that the IS step, at the next iteration, is able to address each sample in $O(1)$ as in (8). This requires redistributing the values in $\mathbf{M}$ according to **ncopies**, which can be done by repeating three analogous phases to *Phase 1*, *Phase 2*, and *Phase 3*. More precisely, we first compute the prefix sum of **ncopies** to obtain $\mathbf{csum} \in \mathbb{N}_0^N$. Then each thread, $id$, searches for a pivot index, $p$, i.e., the first index such that $\mathbf{csum}^p \geq id\frac{N}{P}$. After that, each thread can independently redistribute $\frac{N}{P}$ elements in $\mathbf{M}$ by using Algorithm 1 starting from index $p$.

*Phase 5 - Update* **csumM**. The previous step is computed with a view to updating the values of **csumM**. After *Phase 4*, this can be done by computing (8). For completeness, we note that, having restructured $\mathbf{x}$ to be a flattened 1-D

**Algorithm 3** Novel Parallel Redistribution for Variable Size Samples

**Input:** $\mathbf{x}$, **ncopies**, **csumM**, $N$, $P$, $n = \frac{N}{P}$
**Output:** $\mathbf{x}$, **csumM**

**cdot** $\leftarrow$`Prefix Dot Product`(**ncopies**, $\mathbf{M}$, $P$) //*Phase 1*
Spawn $P$ threads with $id = 0, 1, \ldots, P-1$\{ //Beginning of *Phase 2*
  $W_{\text{tot}} \leftarrow \mathbf{cdot}^{N-1}$
  **if** $id + 1 \leq W_{\text{tot}} \bmod P$ **then**
    $\mathbf{W}^{id} \leftarrow \lfloor \frac{W_{\text{tot}}}{P} \rfloor + 1$, $\mathbf{cw}^{id} \leftarrow id \times \mathbf{W}^{id}$
  **else**
    $\mathbf{W}^{id} \leftarrow \lfloor \frac{W_{\text{tot}}}{P} \rfloor$, $\mathbf{cw}^{id} \leftarrow \lfloor \frac{W_{\text{tot}}}{P} \rfloor \times (\mathbf{W}^{id} + 1) + (id - \lfloor \frac{W_{\text{tot}}}{P} \rfloor) \times \mathbf{W}^{id}$
  **end if**
  $p \leftarrow$`Binary Search` (**cdot**, **ncopies**, $\mathbf{M}$, $\mathbf{cw}^{id}$, $P$) //End of *Phase 2*
  $\mathbf{x}^{n:n\times cp-1} \leftarrow \mathbf{x}^{p}$ //Beginning of *Phase 3*
  $\mathbf{x} \leftarrow$`S-R`$(\mathbf{x}^{p+1:N-1}$, $\mathbf{ncopies}^{p+1:N-1}$, $n - cp)$
} //End of *Phase 3*
**csum** $\leftarrow$`Prefix Sum`(**ncopies**, $P$) //Beginning of *Phase 4*
Spawn $P$ threads with $id = 0, 1, \ldots, P-1$\{
  $p \leftarrow$`Binary Search`(**csum**, **ncopies**, $n$, $P$)
  $cp \leftarrow \min(\mathbf{csum}^{p} - id \times n, n)$
  $\mathbf{M}^{n:n\times cp-1} \leftarrow \mathbf{M}^{p}$
  $\mathbf{M} \leftarrow$`S-R`$(\mathbf{M}^{p+1:N-1}$, $\mathbf{ncopies}^{p+1:N-1}$, $n - cp)$
} //End of *Phase 4*
**csumM** $\leftarrow$`Prefix Sum`($\mathbf{M}$, $P$) //*Phase 5*

array, also means that the IS step is required to update **csumM** as in (8), since the proposal $q(\cdot|\cdot)$ may increase or decrease the size of each and every sample.

Algorithm 3 summarizes the previous steps, and Figure 1 illustrates an example for $N = 8$ and $P = 4$. The following lemma analyzes the time complexity.

**Lemma 1** *Let* $\mathbf{x}$ *be a 1-D array that contains* $N$ *samples, each with size* $\mathbf{M}^i$*, for all* $i = 0, 1, 2, \ldots, N-1$*, which could be different from any* $\mathbf{M}^j$ *with* $j \neq i$*. Let* **ncopies** $\in \mathbf{N}_0^N$ *be an array of integers that complies with (6). Algorithm 3 redistributes the samples in* $\mathbf{x}$ *according to* **ncopies** *in* $O(\hat{M} \log_2 N)$ *time complexity, where* $\hat{M} = \frac{1}{N} \sum_i \mathbf{ncopies}^i \mathbf{M}^i$*, i.e., the average size of new samples.*

*Proof.* To prove Lemma 1, we first analyze the time complexity of each step of Algorithm 3 individually. *Phase 1*, and *Phase 5* can be parallelized by using prefix reduction, which is well known to scale as $O(\frac{N}{P} + \log_2 P)$, with a fast constant time that does not depend on the size of the samples.

In *Phase 2*, each thread performs a BS individually, which takes $O(\log_2 N)$ computations, which also are independent of the sizes of the samples.

*Phase 4* redistributes the elements of $\mathbf{M}$ according to **ncopies**. This can be parallelized in $O(\frac{N}{P} + \log_2 N)$ because it consists of one prefix reduction, one BS, and Algorithm 1 being used to redistribute $\frac{N}{P}$ elements of $\mathbf{M}$. The constant time of this phase will be negligible, since $\mathbf{M}$ is a 1-D array of integers, and also independent of the size of the samples.

In *Phase 3*, each thread performs either $\lfloor \frac{W_{\text{tot}}}{P} \rfloor + 1$ or $\lfloor \frac{W_{\text{tot}}}{P} \rfloor$ memory copies, by Equation (10). Due to Equation (7), $\frac{W_{\text{tot}}}{P}$ in (10) can be rearranged as follows:

$$\frac{W_{\text{tot}}}{P} = \frac{\sum_i \mathbf{ncopies}^i \mathbf{M}^i}{P} = N \times \frac{\sum_i \mathbf{ncopies}^i \mathbf{M}^i}{N \times P} = \frac{N}{P} \times \hat{M}$$

Therefore, as $P \to N$, this splitting strategy is analogous to having each thread work on copying $O(1)$ sample, with size $\hat{M} = \frac{1}{N} \sum_i \mathbf{ncopies}^i \mathbf{M}^i$, i.e., equal to the average size of the new samples.

Overall, all steps combined amount to a total time complexity equal to $O(\hat{M} \log_2 N)$ as $P \to N$. In other words, Algorithm 3 scales logarithmically with $N$ and with a computational cost per sample equal to $O(\hat{M})$. □

**Theorem 2.** *Algorithm 3 divides the workload optimally across the SM threads.*

*Proof.* To prove this theorem, one needs to prove that the time complexity of Algorithm 3, $O(\hat{M} \log_2 N)$ as proven in Lemma 1, is optimal both with respect to number of samples, $N$, and the size of the samples.

With respect to $N$, Algorithm 3 scales logarithmically, i.e., the same time complexity as the prefix sum in (4), which is proven to be optimal [29].

With respect to the sample sizes, each thread in Algorithm 3 performs as many memory copies per sample as in (10). Equation (10) guarantees that the cost of computation per sample performed by each thread is at most within one memory copy of the ideal average, $\frac{W_{\text{tot}}}{P}$. It is then straightforward to infer that Algorithm 3 always achieves the optimal workload balance, as any other strategy which assigns more than $\lfloor \frac{W_{\text{tot}}}{P} \rfloor + 1$ memory copies to any thread, would necessarily underwork/overwork a subset of the threads by at least two memory copies with respect to the ideal average. □

## 5 Numerical Results

In this section, we want to compare two versions of the same parallel SMC sampler sampling from a posterior distribution of variable-size samples: one version utilizing Algorithm 2 to parallelize the redistribution step, and the other version utilizing our proposed approach, i.e., Algorithm 3. For transparency, we note that all codes are implemented in C++ and parallelized with OpenMP 4.5. Implementation details are omitted for brevity due to the page limit, but the reader can access the code from the following link: `https://github.com/AVarsi88/Parallel_SMC_on_Shared_Memory`.

From the results in Section 4, we expect both SMC samplers to scale progressively as we increase the number of threads, $P$. However, given the improved cost of computation per sample, we expect SMC with Algorithm 3 to be much faster than SMC with Algorithm 2 when $\hat{M} \ll \overline{M}$, and we expect the two approaches to achieve similar performance when $\hat{M} \approx \overline{M}$. To test this, we generate samples of variable size by sampling from the following statistical model:

$$\pi(\cdot, \cdot) = \begin{cases} 0.5e^M(1-\alpha)(\mathbf{m}_s^1 - \mathbf{m}_s^0 + 1), \text{ if } \mathbf{m}_s^0 \leq M \leq \mathbf{m}_s^1 \\ 0.5e^M\alpha(\mathbf{m}_b^1 - \mathbf{m}_b^0 + 1), \text{ if } \mathbf{m}_b^0 \leq M \leq \mathbf{m}_b^1. \end{cases} \tag{13}$$

This model is well-suited for our testing purposes because it gives us precise control over the sample sizes and their variability. Indeed, the samples generated by an SMC sampler sampling from (13) will contain a varying number, $M$, of coin flip results (i.e., either 0 or 1). Most importantly, each sample will either have a large size in the range $\mathbf{m}_b$ with a probability $p(M \in \mathbf{m}_b) = \alpha$, or a small size in the range $\mathbf{m}_s$ with a probability $p(M \in \mathbf{m}_s) = 1 - \alpha$. This is illustrated in Figure 2. In our test, we arbitrarily choose $\mathbf{m}_s = \{1, 3\}$ dimensions for the small samples, and a (significantly larger) range of $\mathbf{m}_b = \{698, 700\}$ dimensions for the big samples, such that $\overline{M} = 700$. Then we control the average size of the samples, by varying $\alpha$. More precisely, $\hat{M} \ll \overline{M}$ when $\alpha$ is small, and $\hat{M} \approx \overline{M}$ when $\alpha \to 1.0$. Indeed, we use $\alpha = \{0.01, 0.5, 1.0\}$.

The experimental results are provided in Figures 3, and 4. These results are taken on a workstation that mounts a 2 Xeon Gold 6138 CPU which provides up to 40 physical cores, and 384GB of memory. Therefore, we use power-of-two numbers up to $P = 32$ SM threads for the experiments as both Algorithms 2, and 3 use the divide-and-conquer paradigm. We also use power-of-two numbers for $N$, to make it easier to balance the workload across the SM threads. More precisely, we use $N = \{2^{10}, 2^{15}, 2^{20}\}$. We also note that all reported run-times are average results for 10 Monte Carlo (MC) runs, and, for each MC run, both versions of the SMC samplers are set to perform $K = 10$ iterations.

Figure 3 shows the run-times of both SMC samplers when $\alpha$ increases. As theory claims, when the samples' sizes are roughly constant (i.e., $\alpha = 1.0$) the run-times of the two SMC samplers are comparable, independently of $N$. However, when $\alpha$ decreases, an SMC sampler with Algorithm 3 becomes up to $10\times$ faster than the SMC sampler with Algorithm 2. Due to page limitations, we only report the most informative run-times, i.e., for $P = 32$, as the same run-times for any other $P$ would be qualitatively similar. Figure 4 shows how the run-times of both SMC samplers scale for increasing $P$. As we anticipated, both SMC samplers scale progressively with $P$, but the SMC with Algorithm 3 has faster run-times due to the improved cost of computation per sample.

## 6 Conclusion and Future Work

In this paper, we tackle on SM the Optimal Parallel Redistribution (OPR) problem in the context of SMC samplers drawing $N$ variable-size samples. We first prove that if the samples are assumed to be indivisible (i.e., every core must hold an integer number of samples), OPR is NP-complete. We then prove that, if the indivisibility assumption is relaxed, it is possible to construct a parallel redistribution algorithm that divides the workload optimally between the processing elements, as it achieves optimal $O(\hat{M} \log_2 N)$ time complexity, where $\hat{M}$ is the average size of the redistributed samples. The baseline for comparison is presented in [24], which describes a parallel redistribution algorithm for variable-size, indivisible samples which achieves a sub-optimal $O(\overline{M} \log_2 N)$ time complexity, where $\overline{M}$ is the size of the biggest sample (i.e., the worst-case scenario). The numerical results on a 32-core SM machine show that an SMC sampler equipped

with our proposed parallel redistribution is up to 10× faster than an equivalent SMC sampler equipped with the parallel redistribution in [24].

Despite the encouraging results, several future avenues remain to be explored. First, the run-time of the proposed method could be accelerated by using GPUs instead of CPUs. Another usual practice in supercomputing is to combine DM and SM, e.g., with MPI+X programming models. This would require an effective DM parallel redistribution for variable-size samples. While OPR for variable-size samples on DM is NP-complete (as discussed in this paper), it may still be possible to design an effective (albeit sub-optimal) algorithm. Developing such a redistribution component is, to the best of our knowledge, an open research question.

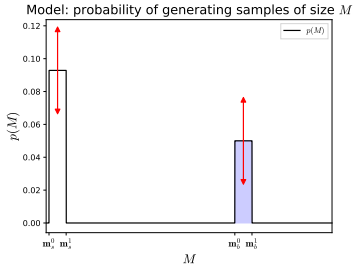
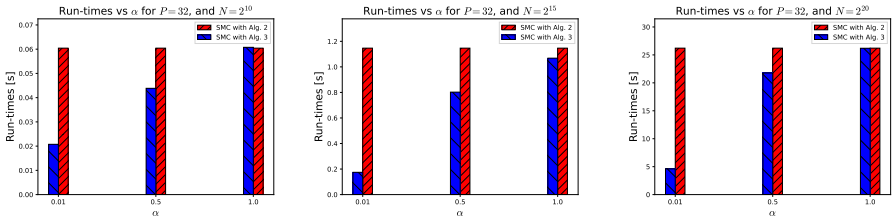
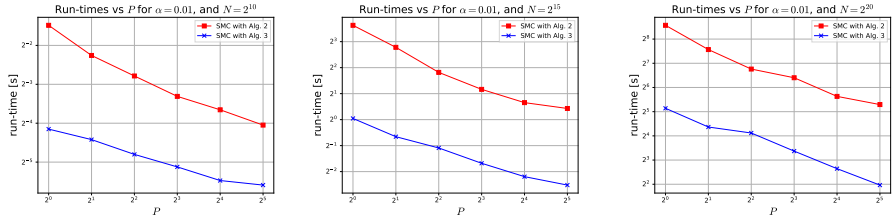



Fig. 2: Model: $p(M)$. The shaded region is $\alpha$, the total probability of drawing big samples. The red arrows symbolize the possibility of having varying $\alpha$.




(a) $N = 2^{10}$ (b) $N = 2^{15}$ (c) $N = 2^{20}$

Fig. 3: Run-times as function of $\alpha$, $N$, and for $P = 32$.




(a) $N = 2^{10}$ (b) $N = 2^{15}$ (c) $N = 2^{20}$

Fig. 4: Run-times as function of $N$, $P$ and for $\alpha = 0.01$.

## References


1. X. Ma, P. Karkus, D. Hsu, and W. S. Lee, "Particle filter recurrent neural networks," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, pp. 5101–5108, Apr. 2020.
2. E. Drousiotis, A. Varsi, P. G. Spirakis, and S. Maskell, "An smc sampler for decision trees with enhanced initial proposal for stochastic metaheuristic optimization," in *Learning and Intelligent Optimization* (P. Festa, D. Ferone, T. Pastore, and O. Pisacane, eds.), (Cham), pp. 123–137, Springer Nature Switzerland, 2025.
3. S. Habibi, E. Drousiotis, A. Varsi, S. Maskell, R. Moore, and P. G. Spirakis, "Conditional importance resampling for an enhanced sequential monte carlo sampler," in *Learning and Intelligent Optimization* (P. Festa, D. Ferone, T. Pastore, and O. Pisacane, eds.), (Cham), pp. 169–184, Springer Nature Switzerland, 2025.
4. M. Barazandegan, F. Ekram, E. Kwok, B. Gopaluni, and A. Tulsyan, "Assessment of type ii diabetes mellitus using irregularly sampled measurements with missing data," *Bioprocess and biosystems engineering*, vol. 38, pp. 615–629, 2015.
5. C. A. Naesseth, F. Lindsten, and T. B. Schön, "High-dimensional filtering using nested sequential monte carlo," *IEEE Transactions on Signal Processing*, vol. 67, pp. 4177–4188, Aug 2019.
6. E. Drousiotis, A. M. Phillips, P. G. Spirakis, and S. Maskell, "Bayesian decision trees inspired from evolutionary algorithms," in *Learning and Intelligent Optimization* (M. Sellmann and K. Tierney, eds.), (Cham), pp. 318–331, Springer International Publishing, 2023.
7. A. Varsi, L. Devlin, P. Horridge, and S. Maskell, "A general-purpose fixed-lag no-u-turn sampler for nonlinear non-gaussian state space models," *IEEE Transactions on Aerospace and Electronic Systems*, pp. 1–16, 2024.
8. A. Svensson, T. B. Schön, and F. Lindsten, "Learning of state-space models with highly informative observations: A tempered sequential monte carlo solution," *Mechanical systems and signal processing*, vol. 104, pp. 915–928, 2018.
9. 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," in *2014 IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, pp. 340–345, July 2014.
10. F. Lopez, L. Zhang, A. Mok, and J. Beaman, "Particle filtering on gpu architectures for manufacturing applications," *Computers in Industry*, vol. 71, pp. 116 – 127, 2015.
11. D. Sheinson, Y. Meng, and D. Elsea, "Pin67 real-time analysis of covid-19 data using sequential monte carlo methods," *Value in Health*, vol. 24, p. S118, 2021.
12. G. D. Ramaraj, S. Venkatakrishnan, G. Balasubramanian, and S. Sridhar, "Aerial surveillance of public areas with autonomous track and follow using image processing," in *2017 International Conference on Computer and Drone Applications (IConDA)*, pp. 92–95, IEEE, 2017.
13. S. Brooks, "Markov chain monte carlo method and its application," *Journal of the royal statistical society: series D (the Statistician)*, vol. 47, no. 1, pp. 69–100, 1998.
14. A. Varsi, S. Maskell, and P. G. Spirakis, "An o (log2n) fully-balanced resampling algorithm for particle filters on distributed memory architectures," *Algorithms*, vol. 14, no. 12, p. 342, 2021.
15. C. Rosato, A. Varsi, J. Murphy, and S. Maskell, "An o(log2 n) smc2 algorithm on distributed memory with an approx. optimal l-kernel," in *2023 IEEE Symposium Sensor Data Fusion and International Conference on Multisensor Fusion and Integration (SDF-MFI)*, pp. 1–8, 2023.

16. L. M. Murray, A. Lee, and P. E. Jacob, "Parallel resampling in the particle filter," *Journal of Computational and Graphical Statistics*, vol. 25, no. 3, pp. 789–805, 2016.
17. 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*, vol. 27, pp. 1570–1574, 2020.
18. B. Lakshminarayanan, D. Roy, and Y. Whye Teh, "Top-down particle filtering for bayesian decision trees," in *Proceedings of the 30th International Conference on Machine Learning* (S. Dasgupta and D. McAllester, eds.), vol. 28 of *Proceedings of Machine Learning Research*, (Atlanta, Georgia, USA), pp. 280–288, PMLR, 17–19 Jun 2013.
19. B. Lakshminarayanan, *Decision trees and forests: a probabilistic perspective*. PhD thesis, UCL (University College London), 2016.
20. N. Quadrianto and Z. Ghahramani, "A very simple safe-bayesian random forest," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 37, no. 6, pp. 1297–1303, 2015.
21. J. Gardner, C. Guo, K. Weinberger, R. Garnett, and R. Grosse, "Discovering and exploiting additive structure for bayesian optimization," in *Artificial Intelligence and Statistics*, pp. 1311–1319, PMLR, 2017.
22. F. Saad, B. Patton, M. D. Hoffman, R. A. Saurous, and V. Mansinghka, "Sequential Monte Carlo learning for time series structure discovery," in *Proceedings of the 40th International Conference on Machine Learning* (A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, eds.), vol. 202 of *Proceedings of Machine Learning Research*, pp. 29473–29489, PMLR, 23–29 Jul 2023.
23. E. Drousiotis, P. G. Spirakis, and S. Maskell, "Parallel approaches to accelerate bayesian decision trees," *arXiv preprint arXiv:2301.09090*, 2023.
24. E. Drousiotis, A. Varsi, P. G. Spirakis, and S. Maskell, "A shared memory smc sampler for decision trees," in *2023 IEEE 35th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 209–218, 2023.
25. E. Drousiotis, A. Varsi, A. M. Phillips, S. Maskell, and P. G. Spirakis, "A massively parallel smc sampler for decision trees," *Algorithms*, vol. 18, no. 1, 2025.
26. P. D. Moral, A. Doucet, and A. Jasra, "Sequential monte carlo samplers," *Journal of the Royal Statistical Society. Series B (Statistical Methodology)*, vol. 68, no. 3, pp. 411–436, 2006.
27. J. D. Hol, T. B. Schon, and F. Gustafsson, "On resampling algorithms for particle filters," in *2006 IEEE Nonlinear Statistical Signal Processing Workshop*, pp. 79–82, Sept 2006.
28. R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *Journal of the ACM*, vol. 27, p. 831–838, Oct. 1980.
29. E. E. Santos, "Optimal and efficient algorithms for summing and prefix summing on parallel machines," *Journal of Parallel and Distributed Computing*, vol. 62, no. 4, pp. 517–543, 2002.
30. S. Kozakai, N. Fujimoto, and K. Wada, "Efficient gpu-implementation for integer sorting based on histogram and prefix-sums," in *Proceedings of the 50th International Conference on Parallel Processing*, ICPP '21, (New York, NY, USA), Association for Computing Machinery, 2021.
31. M. R. Garey and D. S. Johnson, *Computers and intractability: A Guide to the Theory of NP-Completeness*, vol. 174. freeman San Francisco, 1979.