A Shared Memory SMC Sampler for Decision Trees

Efthyvoulos Drousiotis
Department of Electrical Engineering and Electronics
University of Liverpool
Liverpool L69 3BX, UK
E.Drousiotis@liverpool.ac.uk

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

Paul G. Spirakis
Department of Computer Science
University of Liverpool
Liverpool L69 3BX, UK
P.Spirakis@liverpool.ac.uk

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

***Abstract*—Modern classification problems tackled by using Decision Tree (DT) models often require demanding constraints in terms of accuracy and scalability. This is often hard to achieve due to the ever-increasing volume of data used for training and testing. Bayesian approaches to DTs using Markov Chain Monte Carlo (MCMC) methods have demonstrated great accuracy in a wide range of applications. However, the inherently sequential nature of MCMC makes it unsuitable to meet both accuracy and scaling constraints. One could run multiple MCMC chains in an embarrassingly parallel fashion. Despite the improved run-time, this approach sacrifices accuracy in exchange for strong scaling. Sequential Monte Carlo (SMC) samplers are another class of Bayesian inference methods that also have the appealing property of being parallelizable without trading off accuracy. Nevertheless, finding an effective parallelization for the SMC sampler is difficult, due to the challenges in parallelizing its bottleneck, redistribution, in such a way that the workload is equally divided across the processing elements, especially when dealing with variable-size models such as DTs. This study presents a parallel SMC sampler for DTs on Shared Memory (SM) architectures, with an $O(\log_2 N)$ parallel redistribution for variable-size samples. On an SM machine mounting 32 cores, the experimental results show that our proposed method scales up to a factor of 16 compared to its serial implementation, and provides comparable accuracy to MCMC, but 51 times faster.**

***Index Terms*—Parallel Algorithms, Sequential Monte Carlo Samplers, Markov Chain Monte Carlo, Bayesian Decision Trees, Shared Memory Programming.**

I. Introduction

A. Motivation

Decision Tree (DT) models are well-used algorithms to solve classification problems in the field of Machine Learning (ML). Its application domain is vast and diverse since it ranges from medicine [1] to biology [2], chemistry [3], and engineering [4]. While real-world applications provide access to large amounts of data, this often translates into challenging accuracy and run-time constraints.

In the recent past, Markov Chain Monte Carlo (MCMC) methods have emerged as a popular Bayesian approach to DTs [5]–[8], often providing better classification accuracy than traditional classification methods, such as decision forests [9], [10]. Despite its state-of-the-art accuracy performance, MCMC is also highly computationally intensive. This limitation would typically be compensated by using parallel computing on multi-core architectures. However, MCMC is widely-known to be inherently sequential, making it challenging to tackle modern classification problems both accurately and quickly. Therefore, we argue that a Bayesian alternative to MCMC for DTs to solve ML classification problems, which is both parallelizable and accurate, would be greatly desirable.

B. Related Work

Bayesian DTs for classification problems in ML were first presented in [11], [12] and improved in [13], [14]. The idea is to randomly generate a chain of N samples (each sample being a DT) in order to estimate the possible problem outcomes. This approach is used extensively [15], [16], given the good performance in terms of classification accuracy, but does not involve any parallel components in its implementation.

Each new sample in an MCMC chain is generated given the previous one, which is why this approach is hard to perform in parallel. There exist several attempts to parallelize a single MCMC chain [17], [18]; examples can also be found in the context of DTs [19]. These approaches are strongly problem specific and, as such, do not lead to good scalability in most cases. Another widely used parallelization strategy is to run multiple chains in parallel. There are numerous examples of this method in the literature in the context of DTs [20]–[22]. However, this parallelization strategy trades off the accuracy in order to achieve good scaling and run-time performance.

Sequential Monte Carlo (SMC) samplers [23] are another class of Monte Carlo (MC) methods that can be used in the same context as MCMC. The overall idea is to use a combination of sampling and resampling to generate a population of N random samples. The advantage of SMC is that it can provide competitive accuracy but is also parallelizable, because the samples are generated independently. However, an effective parallelization of SMC is not straightforwardly achievable, given the challenges involved in parallelizing the bottleneck, redistribution (a sub-task of resampling). The work in [24] describes a parallelization of the redistribution step for Distributed Memory (DM) architectures, while [25]–[28] focus on Shared Memory Programming (SMP). These parallelization strategies are specific for the case where all samples have fixed sizes. However, in the case of DTs, as well as other Abstract Data Types (ADTs) such as additive structures [29], the samples have variable sizes, meaning that the approaches in [24]–[28] are not straightforwardly applicable. In [30], a parallel SMC sampler for DTs on SMP is presented, but only the sampling step is parallelized, while resampling is executed sequentially due to the challenges in parallelizing redistribution, which practically resulted in no scalability. Another similar method to a conventional SMC sampler for DTs is found in [31]. This approach is implemented sequentially, and in [32] is shown to provide significantly lower accuracy than MCMC.

C. Contribution and Paper Outline

In this paper, we present a parallel implementation of an SMC sampler for DTs (and, as such, for application domains that fall in the scope of ML classification problems) on SMP that is fully parallelized in all its components. In doing so, we describe a parallelization for the redistribution step which works for any variable-size samples, including DTs, and achieves asymptotically optimal $O(\log_2 N)$ time complexity. On a Shared Memory (SM) machine running 32 parallel threads, our proposed method achieves up to a 16-fold speed-up vs its sequential implementation and offers a better accuracy-vs-run-time compromise than MCMC. More precisely, our approach provides comparable accuracy to MCMC but up to 51 times faster given the same problem size, or significantly better accuracy for the same run-time.

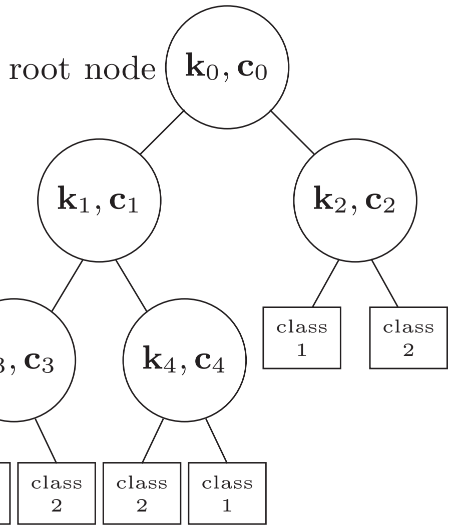
The remainder of this paper is organized as follows: Section II briefly describes the DT model. Section III gives details about SMC and MCMC, with a view to emphasizing their implementation details in the context of DTs. Section IV presents and analyzes our approach in details. Section V shows the experimental results for two commonly used classification datasets. Section VI draws the final conclusions and gives suggestions for future work.

II. Decision Trees

In this section, we briefly describe the anatomy of a DT, in all its components, and how this model is used for classification tasks. The reader is referred to [33] for further details.

Given a dataset comprising l records, $\mathbf{Y} \in \mathbb{Z}^l$, and corresponding matrix of features, $\mathbf{x} \in \mathbb{R}^{l \times r}$, the DT model is trained to classify a datum $\mathbf{Y}^j$ given the corresponding j-th row of features in $\mathbf{x}$.

For a given tree, $\mathbf{T}$, we define $d(\mathbf{T})$ to be the depth of the tree, $D(\mathbf{T})$ to be the set of the m non-leaf nodes, and $L(\mathbf{T})$ to be the set of leaf nodes. The tree, $\mathbf{T}$, is then parameterized by the set of features for all non-leaf nodes, $\mathbf{k}$, and the vector of corresponding thresholds, $\mathbf{c}$. A DT operates by descending a tree. The process of outputting a classification outcome for



root node
$\mathbf{k}_0, \mathbf{c}_0$
$\mathbf{k}_1, \mathbf{c}_1$
$\mathbf{k}_2, \mathbf{c}_2$
$\mathbf{k}_3, \mathbf{c}_3$
$\mathbf{k}_4, \mathbf{c}_4$
class 1
class 2
class 1
class 2
class 2
class 1

Fig. 1: An example of a DT with $d(\mathbf{T}) = 3$, $m = 5$, and six leaf nodes with a quantitative response of two classes.

a given datum starts at the root node. At each non-leaf node, a decision as to which child node to progress to is made based on the datum and the parameters of the node. This process continues until a leaf node is reached. At the leaf node, a node-specific and datum-independent classification output is generated. Figure 1 exemplifies a DT with five non-leaf nodes and six leaf nodes.

III. Bayesian Decision Trees

MC methods can be used to make estimations of the state of a statistical model, which is described by a posterior distribution (or simply posterior, the terms are used interchangeably), i.e., the probability density function of the state of the model given some data. The idea is to generate a population of random samples that represent the posterior, with a view to using the samples to make estimates. However, posterior distributions are often challenging to be sampled from directly. MCMC and SMC are two of the most popular Bayesian classes of methodologies for sampling from a posterior distribution.

When it comes to Bayesian approaches to DTs, we are interested in making estimations of the true tree, $\mathbf{T}$, and its set of features, θ, given some data, $\mathbf{Y}$, and its features, $\mathbf{x}$, for a classification problem. Therefore, $\pi(\mathbf{T}, \theta | \mathbf{Y}, \mathbf{x})$ is the posterior from which we want to generate random samples.

In this section, we describe how both MCMC and SMC can be implemented in the specific case of sampling a population of DTs. The reader is referred to [11], [12], [23].

A. Markov Chain Monte Carlo

The idea behind MCMC methods is to perform N_T iterations, $\forall k = 0, 1, 2, \dots, N_T - 1$, each of which generates a sample, $\mathbf{t}_k$, of $\pi(\mathbf{T}, \theta | \mathbf{Y}, \mathbf{x})$, each sample including a possible guess of $\mathbf{T}$ and θ. The first sample, $\mathbf{t}_0$, is generated from an arbitrary initial proposal distribution (or initial proposal, the terms are used interchangeably), $q_0(\cdot)$, from which it is easy to sample. After that, each iteration, $\forall k$, first proposes a new sample of the posterior by sampling from another proposal distribution (or proposal, the terms are used interchangeably), $q(\cdot|\mathbf{t}_{k-1})$, as follows:

$$\mathbf{t}' \sim q(\cdot|\mathbf{t}_{k-1}), \tag{1}$$

where the proposal is also arbitrary, and conveniently easy to sample from.

The proposed sample, $\mathbf{t}'$, will then be either accepted or rejected according to the following acceptance probability:

$$a(\mathbf{t}', \mathbf{t}_{k-1}) = \min\left(1, \frac{\pi\left(\mathbf{t}'|\mathbf{Y}, \mathbf{x}\right)}{\pi\left(\mathbf{t}_{k-1}|\mathbf{Y}, \mathbf{x}\right)} \frac{q\left(\mathbf{t}_{k-1}|\mathbf{t}'\right)}{q\left(\mathbf{t}'|\mathbf{t}_{k-1}\right)}\right). \tag{2}$$

In other words, the new sample, $\mathbf{t}_k$, will be:

$$\mathbf{t}_k = \begin{cases} \mathbf{t}' & \text{if } u \leq a(\mathbf{t}', \mathbf{t}_{k-1}), \\ \mathbf{t}_{k-1} & \text{if } u > a(\mathbf{t}', \mathbf{t}_{k-1}), \end{cases} \tag{3}$$

where $u \sim \texttt{Uniform}[0, 1]$.

After $N_T - 1$ iterations, we have a single chain of N_T samples from which we can estimate $\mathbf{T}$ and θ, by either computing the mean of the samples or using majority voting. However, before doing that, an initial subset of the N_T samples, typically $N_b = 0.5N_T$, is discarded, a process called burn-in. This is done to improve the quality of the estimation, as the first samples are often not converged. In Section V, we refer to this method (which is also illustrated in Algorithm 1) as Single-Chain MCMC.

Algorithm 1 Single-Chain MCMC

```
Input: $N_T, N_b$
Output: $\mathbf{t}$
 1: $\mathbf{t}_0 \sim q_0(\cdot)$
 2: for $k \leftarrow 1$; $k < N_T$; $k \leftarrow k + 1$ do
 3:     $\mathbf{t}' \sim q(\cdot|\mathbf{t}_{k-1})$
 4:     $a(\mathbf{t}', \mathbf{t}_{k-1}) \leftarrow \min\left(1, \frac{\pi(\mathbf{t}'|\mathbf{Y},\mathbf{x})}{\pi(\mathbf{t}_{k-1}|\mathbf{Y},\mathbf{x})} \frac{q(\mathbf{t}_{k-1}|\mathbf{t}')}{q(\mathbf{t}'|\mathbf{t}_{k-1})}\right)$
 5:     $u \sim \texttt{Uniform}[0, 1]$
 6:     if $u \leq a(\mathbf{t}', \mathbf{t}_{k-1})$ then
 7:         $\mathbf{t}_k \leftarrow \mathbf{t}'$
 8:     else
 9:         $\mathbf{t}_k \leftarrow \mathbf{t}_{k-1}$
10:     end if
11: end for
12: Discard first $N_b$ samples in $\mathbf{t}$ $\forall k = 0, 1, \ldots, N_b - 1$
```

This approach is inherently sequential, as each sample is generated given the previous one. One could try to parallelize either (1) or (2), as done in [19], for example. However, this approach is strongly model dependent and often results in poor scalability. Another alternative is to run N chains in parallel, each with K samples [20], [22], such that the total number of samples is $N_T = N \times K$. In Section V, we refer to this method (which is briefly summarized in Algorithm 2) as Multi-Chain MCMC. Since the chains are generated in an embarrassingly parallel fashion, Algorithm 2 scales as $O(\frac{N}{P})$ if using P processing elements.

Algorithm 2 Multi-Chain MCMC

```
Input: $N, K$
Output: $\mathbf{t}$
 1: $N_b \leftarrow \frac{K}{2}$
 2: for $i \leftarrow 0$; $i < N$; $i \leftarrow i + 1$ do in parallel
 3:     $\mathbf{t}^i_{0:K-N_b-1} \leftarrow$ Single-Chain MCMC$(K, N_b)$
 4: end for
```

B. Sequential Monte Carlo

There exist several variants of SMC samplers. However, the general idea is to perform sampling and resampling in sequence for K iterations, $\forall k = 0, 1, \ldots, K-1$, each of which will generate and update N independent, weighted samples, $\mathbf{t}^i_k$, $\forall i = 0, 1, 2, \ldots, N-1$. In this paper, we use Sampling Using Multinomial Distribution (SUMD), a variant of SMC that revisits some concepts from MCMC. The samples are first initialized from the initial proposal as follows:

$$\mathbf{t}^i_0 \sim q_0(\cdot), \quad \forall i = 0, 1, 2, \ldots, N-1. \tag{4}$$

Then, during each SMC iteration, $\forall k = 1, 2, \ldots, K-1$, the samples are updated from the proposal and weighted as follows:

$$\mathbf{t}^i_k \sim q(\cdot|\mathbf{t}^i_{k-1}), \quad \forall i = 0, 1, 2, \ldots, N-1, \tag{5a}$$

$$\mathbf{w}^i_k = \frac{\pi\left(\mathbf{t}^i_k|\mathbf{Y}, \mathbf{x}\right)}{\pi\left(\mathbf{t}^i_{k-1}|\mathbf{Y}, \mathbf{x}\right)} \frac{q\left(\mathbf{t}^i_{k-1}|\mathbf{t}^i_k\right)}{q\left(\mathbf{t}^i_k|\mathbf{t}^i_{k-1}\right)}, \quad \forall i = 0, 1, 2, \ldots, N-1. \tag{5b}$$

The weights $\mathbf{w}^i_k$, $\forall i = 0, 1, \ldots, N-1$, are then normalized such that $\sum_i \tilde{\mathbf{w}}^i_k = 1$, i.e., 100% of the total probability space. The normalized weights are then computed as follows:

$$\tilde{\mathbf{w}}^i_k = \frac{\mathbf{w}^i_k}{\sum_{j=0}^{N-1} \mathbf{w}^j_k}, \quad \forall i = 0, 1, 2, \ldots, N-1. \tag{6}$$

Since the samples are generated from $q(\cdot|\cdot)$, and not directly from the posterior distribution of interest, this sampling strategy may suffer from a numerical error, called degeneracy, which could make most of the weights equal to 0, leading towards poor estimations of the state. This problem is commonly tackled by using resampling, an algorithm that overwrites the samples with low weights with copies of the samples with high weights. Several variants of resampling can be found in the literature [34]. SUMD uses multinomial resampling, which performs two steps.

In the first step, we compute $\mathbf{ncopies} \in \mathbb{Z}^N$, an array of N non-negative integers whose i-th element, $\mathbf{ncopies}^i$, says how many copies of the i-th sample, $\mathbf{t}^i_k$, are necessary to keep. To do that, multinomial resampling first computes $\mathbf{cdf} \in \mathbb{R}^N$, the cumulative sum (or prefix sum, the terms are used interchangeably) of $\tilde{\mathbf{w}}$, as follows:

$$\mathbf{cdf}^i = \sum_{j=0}^{i-1} \tilde{\mathbf{w}}^j_k, \quad \forall i = 0, 1, 2, \ldots, N-1. \tag{7}$$

Then, $\mathbf{ncopies}^i$ is computed by

$$\mathbf{ncopies}^i = \lceil \mathbf{cdf}^{i+1} - u \rceil - \lceil \mathbf{cdf}^i - u \rceil, \quad \forall i = 0, 1, \ldots, N-1, \tag{8}$$

where $u \sim$Uniform$[0,1]$, and $\lceil\cdot\rceil$ is the ceiling operator. From (7) and (8) it is relatively straightforward to infer that

$$0 \leq \mathbf{ncopies}^i \leq N, \quad \forall i = 0,1,2,\dots,N-1, \tag{9a}$$

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

The second step is redistribution, which creates a new population of samples by duplicating each sample, $\mathbf{t}_k^i$, a total of $\mathbf{ncopies}^i$ times, meaning that those samples for which $\mathbf{ncopies}^i = 0$ will be deleted. Algorithm 3 illustrates a possible implementation of a Sequential Redistribution (S-R).

Algorithm 3 Sequential Redistribution (S-R)

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

After resampling, a new iteration starts from (5) and, after $K-1$ iterations, the final samples are used to make estimates, by either computing the mean, or the majority voting. Algorithm 4 summarizes the steps above. In line with the notation used for Single-Chain MCMC and Multi-Chain MCMC, we say that SMC has a total problem size equal to $N_T = N \times K$, since each of the N samples is updated K times.

Algorithm 4 Sampling Using Multinomial Distribution

```
Input: $K$, $N$
Output: $\mathbf{t}_k$
    $\mathbf{t}_0^i \sim q_0(), \forall i = 0,1,\dots,N-1$
    for $k \leftarrow 0$; $k < K$; $k \leftarrow k+1$ do
        $\mathbf{t}_k^i \sim q(\cdot|\mathbf{t}_{k-1}^i), \quad \forall i = 0,1,2,\dots,N-1$
        $\mathbf{w}_k^i \leftarrow \frac{\pi(\mathbf{t}_k^i|\mathbf{Y},\mathbf{x})}{\pi(\mathbf{t}_{k-1}^i|\mathbf{Y},\mathbf{x})} \frac{q(\mathbf{t}_{k-1}^i|\mathbf{t}_k^i)}{q(\mathbf{t}_k^i|\mathbf{t}_{k-1}^i)}, \quad \forall i = 0,1,2,\dots,N-1$
        $\tilde{\mathbf{w}}_k^i \leftarrow \frac{\mathbf{w}_k^i}{\sum_{j=0}^{N-1}\mathbf{w}_k^j}, \quad \forall i = 0,1,2,\dots,N-1$
        $\mathbf{cdf}^i \leftarrow \sum_{j=0}^{i-1}\tilde{\mathbf{w}}_k^j, \quad \forall i = 0,1,\dots,N-1$
        $u \sim$Uniform$[0,1]$
        $\mathbf{ncopies}^i \leftarrow \lceil\mathbf{cdf}^{i+1}-u\rceil-\lceil\mathbf{cdf}^i-u\rceil, \ \forall i = 0,1,\dots,N-1$
        $\mathbf{t}_k \leftarrow$Redistribution$(\mathbf{t}_k, \mathbf{ncopies}, N)$
    end for
```

C. Posterior and Proposal for Bayesian Decision Trees

In this section, we briefly describe how to compute the posterior, the proposal, and the initial proposal in the specific case of DTs.

1) Posterior Distribution: The posterior is proportional (up to a normalization constant) to the product of the likelihood and the prior. In other words:

$$\begin{aligned}\pi(\mathbf{T},\theta|\mathbf{Y},\mathbf{x}) &\propto p(\mathbf{Y}|\mathbf{T},\theta,\mathbf{x})p(\theta,\mathbf{T}) \\ &= p(\mathbf{Y}|\mathbf{T},\theta,\mathbf{x})p(\theta|\mathbf{T})p(\mathbf{T}),\end{aligned} \tag{10}$$

where $p(\mathbf{Y}|\mathbf{T},\theta,\mathbf{x})$ is the likelihood and $p(\theta,\mathbf{T})$ is the prior. More precisely, for each individual term in (10), we use the following expressions:

$$p(\mathbf{Y}|\mathbf{T},\theta,\mathbf{x}) = \prod_{j=0}^{l-1} p(\mathbf{Y}_j|\mathbf{x}_j,\mathbf{T},\theta), \tag{11}$$

$$p(\theta|\mathbf{T}) = \prod_{j=0}^{m-1} p(\mathbf{k}_j|\mathbf{T})p(\mathbf{c}_j|\mathbf{k}_j,\mathbf{T}), \tag{12}$$

$$p(\mathbf{T}) = \frac{\lambda^m}{(e^\lambda - 1)m!}. \tag{13}$$

Using the Poisson distribution for the $p(\mathbf{T})$ term is a common practice in the literature [12], [15], [16]. As we can see, (13) only requires tuning of the λ hyperparameter. Another valid expression can be found in [35], [36]. This alternative, however, requires tuning of two hyperparameters. Therefore, we employ (13) for simplicity.

2) Proposal Distribution: In the case of the initial proposal distribution, $q_0(\cdot)$, we sample from the prior of the tree, $p(\mathbf{T})$.

The proposal distribution, $q(\cdot|\cdot)$, consists of a set of four possible moves, which may either change the dimensionality of the tree or update its nodes. The possible moves are:

- *Grow*. This move increases the dimensionality of the tree by uniformly selecting one of the leaf nodes and appending two new child nodes to the selected leaf.
- *Prune*. This move decreases the dimensionality of the tree by uniformly selecting one of the non-leaf nodes and removing its child nodes.
- *Change*. This move picks uniformly a non-leaf node, j, and changes its feature, $\mathbf{k}^j$, and threshold, $\mathbf{c}^j$. In other words, this move neither increases nor decreases the dimensionality of the tree.
- *Swap*. This move picks two non-leaf nodes, i and j, uniformly, and swaps their features, $\mathbf{k}^i$ and $\mathbf{k}^j$, and their thresholds, $\mathbf{c}^i$ and $\mathbf{c}^j$. Therefore, this move also maintains the dimensionality of the tree unchanged.

The probabilities of selecting any of these four moves, $p(Grow)$, $p(Prune)$, $p(Change)$, and $p(Swap)$, are user-defined, but they must sum up to 1, and a typical choice is to make these probabilities equal to 25%. The value of $q(\cdot|\cdot)$ to be used in (2) and (5b) is described in detail in [19], and omitted here for brevity.

IV. Parallel Sequential Monte Carlo

In this section, we describe the parallelization of all tasks in the SMC sampler, including our approach to parallelizing the redistribution step, and also give some brief implementation details. All code we have used in this work is implemented in C++ with OpenMP 4.5.

It is relatively straightforward to infer that both steps to initialize and update the samples and the weights, i.e., Equations (4) and (5), are embarrassingly parallel. On SMP (as well as on DM), these steps can be parallelized by equally dividing the iteration space of the related `for` loops, $\forall i$, across the P SM threads, such that each thread, $id = 0,1,\dots,P-1$, works on a chunk of $n = \frac{N}{P}$ samples and weights with index

$i = id \cdot n, id \cdot n+1, id \cdot n+2, \dots, (id+1) \cdot n - 1$. Therefore, these steps scale as $O(\frac{N}{P})$. The same can be said about (8). On OpenMP, embarrassingly parallel tasks are parallelized by adding `#pragma omp parallel for` instructions on top of the related `for` loops.

Equations (6), requires the computation of a vector sum. This operation is parallelizable by using reduction, which notoriously achieves $O(\frac{N}{P} + \log_2 P)$ time complexity. On OpenMP, reducible operators are parallelized by adding the `reduction` clause to the pragma instructions, and specifying the operator (e.g. + for the vector sum and `max` for the vector max) and the variable to reduce.

The prefix sum in Equation (7) can also be performed in $O(\frac{N}{P} + \log_2 P)$ by using prefix reduction. Implementation details are omitted for brevity, but further information can be found in [37], [38].

Algorithm 3 takes $O(N)$ steps on a single core, just like all the other tasks in the SMC sampler when $P = 1$. This is because Equation (9b) holds. However, this algorithm is impossible to be parallelized if using embarrassingly parallel approaches. The main reason is that the workload associated with duplicating each sample, $\mathbf{t}^i$, a total of $\mathbf{ncopies}^i$ times is inherently unbalanced as Equation (9a) holds. In several cases, such as when sampling DTs, the samples have variable sizes, which would potentially make an embarrassingly parallel attempt to parallelize S-R even more unbalanced. Here, we describe an approach that works for variable-size samples, scales as $O(\frac{N}{P} + \log_2 N)$, and maintains the workload on each thread balanced.

Step 1 - Max. Let $\mathbf{M}^i \in \mathbb{Z}^+$, $\forall i = 0, 1, \dots, N-1$, be the size of each sample, $\mathbf{t}^i$. In this step, the threads compute in parallel

$$\overline{M} = \max_{0 \leq i \leq N-1} \mathbf{M}^i, \tag{14}$$

i.e., the size of the biggest sample.

Step 2 - Pad. Each thread, $id = 0, 1, \dots, P-1$, extends each sample by appending $\overline{M} - \mathbf{M}^i$ dimensions each with a value that is known to be impossible. In the specific case of DTs, we can use negative integers such as -1, but we advocate that Not a Numbers (NaNs) would be a generic value for any variable-size ADT. After this step, the samples have all the same size $\overline{M}$.

Step 3 - Prefix Sum. In this step, the threads first compute in parallel $\mathbf{csum} \in \mathbf{Z}^N$, the cumulative sum of $\mathbf{ncopies}$, such that

$$\mathbf{csum}^i = \sum_{j=0}^{i-1} \mathbf{ncopies}^j, \quad \forall i = 0, 1, \dots, N-1. \tag{15}$$

The integer $\mathbf{csum}^i$ represents the number of copies to be created up to the index i. Alternatively, because the particles have now equal sizes, one could think of $\mathbf{csum}^i$ as the total workload (up to a constant time term equal $\overline{M}$) in order to sequentially redistribute the particles from the index 0 to the index i.

Step 4 - Binary Search. It is now possible to perfectly divide the total workload to redistribute N samples between P threads if each thread, $id = 0, 1, \dots, P-1$, searches for an index, called pivot, p, which is the first index that satisfies the following boolean expression:

$$\mathbf{csum}^p \geq id \times n. \tag{16}$$

Since $\mathbf{csum}$ is inherently monotonically increasing, it is also sorted by definition. Therefore, each thread can independently search for its pivot by using Binary Search (BS).

Step 5 - Copy. After *Step 4*, each thread can freely and independently redistribute $\frac{N}{P}$ samples starting from its pivot by using Algorithm 3. However, more than one thread may happen to share the same pivot. Therefore, before using S-R, each thread must first figure out how many copies of $\mathbf{t}^p$ is allowed to create. This is always computed as the

$$\min\left(\mathbf{csum}^p - id \times n, n\right). \tag{17}$$

Indeed, since the workload is divided according to $\mathbf{csum}$ and Equation (16), if two or more threads share the same pivot, only the thread with the highest id must create less than $n = \frac{N}{P}$ copies of $\mathbf{t}^p$. We note that, since this algorithm is designed to run on SMP, in this step it is strongly recommended to use a temporary array where the samples get temporarily copied to, such that the parallel threads will not risk overwriting sensible information.

Step 6 - Restore. Each thread, $id = 0, 1, \dots, P-1$, loops over the samples $\mathbf{t}^i$, $\forall i = id \cdot n, id \cdot n+1, id \cdot n+2, \dots, (id+1) \cdot n-1$, and removes up to $\overline{M}-1$ dimensions from each sample, i.e., those dimensions that are encoded with an impossible value during *Pad*, such as -1s for DTs or NaNs for any ADT.

Algorithm 5 Parallel Redistribution for Variable Size Samples

```
Input: $\mathbf{t}$, $\mathbf{ncopies}$, $\mathbf{M}$, $N$, $P$, $n = \frac{N}{P}$
Output: $\mathbf{t}$
 1: $\overline{M} \leftarrow$ Max($\mathbf{M}$, $P$), spawns & runs $P$ threads
 2: $\mathbf{t} \leftarrow$ Pad($\mathbf{t}$, $\overline{M}$, $P$), spawns & runs $P$ threads
 3: $\mathbf{csum} \leftarrow$ Prefix Sum($\mathbf{ncopies}$, $P$), spawns & runs $P$ threads
 4: Spawn $P$ threads with $id = 0, 1, \dots, P-1${
 5:     $p \leftarrow$ Binary Search($\mathbf{csum}$, $\mathbf{ncopies}$, $n$)
 6:     $cp \leftarrow \min(\mathbf{csum}^p - id \times n, n)$
 7:     $\mathbf{t}_{temp}^{n:n \times cp-1} \leftarrow \mathbf{t}^p$
 8:     $\mathbf{t}_{temp} \leftarrow$ S-R($\mathbf{t}^{p+1:N-1}$, $\mathbf{ncopies}^{p+1:N-1}$, $n - cp$)
 9: }
10: $\mathbf{t} \leftarrow$ Restore($\mathbf{t}_{temp}$, $\overline{M}$, $P$), spawns & runs $P$ threads
```

These steps are summarized in Algorithm 5, and Figure 2 illustrates a practical example for $N = 8$ samples and $P = 4$ threads. In the following theorem and corollary, we analyze the time complexity of our approach.

Theorem IV.1. *Let* $\mathbf{t}$ *be an array of* N *lists, where each list,* $\mathbf{t}^i, \forall i = 0, 1, \dots, N-1$*, has variable size,* $\mathbf{M}^i$*. Let* $\mathbf{ncopies} \in \mathbf{Z}^N$ *be an array of non-negative integers for which (9) holds. On an SM architecture running* P *parallel threads, Algorithm 5 redistributes* $\mathbf{t}$ *according to* $\mathbf{ncopies}$ *in* $O(\frac{N}{P} + \log_2 N)$ *steps with an* $O(\overline{M})$ *constant time factor for each sample.*

Proof. To prove Theorem IV.1, we start by analyzing the time complexity of each of the steps in Algorithm 5 individually.

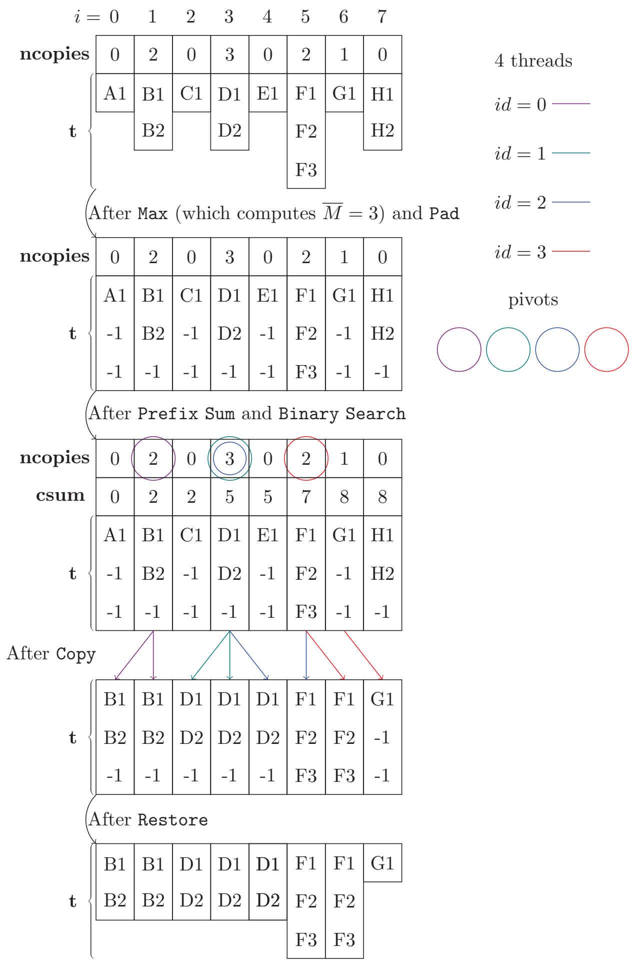


Fig. 2: Parallel Redistribution - Example for $N = 8$ and $P = 4$. Each tree node is encoded with a letter and number for brevity. The -1s represent fake tree nodes.

The computation of $\overline{M}$ in (14) can be parallelized by using parallel reduction, as the vector max operator is also reducible, just like vector sum. Therefore, (14) scales as $O(\frac{N}{P} + \log_2 P)$ with a $O(1)$ constant time factor.

Step 2 and *Step 6* are embarrassingly parallel, as they loop independently over the samples and extend or reduce the dimensionality of each sample by up to $\overline{M} - 1$. Therefore, these two steps scale as $O(\overline{M}\frac{N}{P})$, or, more simply, as $O(\frac{N}{P})$ with a constant time factor per sample of $O(\overline{M})$.

In *Step 3*, the parallel threads compute the cumulative sum of **ncopies**, which, as we said above, achieves $O(\frac{N}{P} + \log_2 P)$ time complexity with a $O(1)$ constant time factor.

In *Step 4*, the threads perform one independent BS each, which notoriously takes $O(\log_2 N)$ comparisons. In addition, each thread also computes (16) once, which takes $O(1)$.

The copying procedure described in *Step 5* requires each thread to copy independently $\frac{N}{P}$ particles, each of which has $\overline{M}$ dimensions. Therefore, this step, just like *Step 2* and *Step 6*, scales as $O(\overline{M}\frac{N}{P})$, or, more simply, as $O(\frac{N}{P})$ with an $O(\overline{M})$ constant time factor per sample.

Hence, by summing up the individual time complexity terms described above, we can conclude that Algorithm 5 runs in $O(\frac{N}{P} + \log_2 N)$ with an $O(\overline{M})$ computational cost on each sample. □

Corollary 1. *Let* $\mathbf{t}$ *be an array of* N *lists, where each list,* $\mathbf{t}^i$, $\forall i = 0, 1, \ldots, N-1$, *has variable size,* $\mathbf{M}^i$, *and normalized weight* $\tilde{\mathbf{w}}^i \in \mathbf{R}$. *On an SM architecture running* P *parallel threads, a resampling algorithm performing steps (7), (8), and Algorithm 5 achieves asymptotically optimal time complexity equal to* $O(\log_2 N)$.

Proof. As mentioned in this section, Equation (8) is embarrassingly parallel, and hence takes $O(1)$ iterations for $P = N$, and Equation (7) can be executed in $O(\log_2 N)$ for $P = N$ by using parallel prefix sum. Furthermore, the time complexity of parallel prefix sum is optimally bound to $O(\log_2 N)$ as proven in [39]. Therefore, given the result from Theorem IV.1, this corollary is straightforwardly proven by absurdity. Indeed, even if it was possible to redistribute N samples in less than $O(\log_2 N)$, such as $O(\log(\log_2 N))$ or $O(1)$, or entirely avoid redistribution, the overall time complexity of resampling would still be bound to $O(\log_2 N)$ due to Equation (7). □

Remark 1. *We note that, although Theorem IV.1 and Corollary 1 prove that the presented approach achieves the asymptotic lower bound (which straightforwardly makes Algorithm 4 achieve the optimal time complexity as well), the constant time per sample in the resampling algorithm is bound to* $O(\overline{M})$, *which is not optimal. In fact, in this work, we have employed static load balancing to parallelize Algorithm 3. However, dynamic load balancing solutions could be an interesting alternative to explore in future work because, despite often showing data-dependent performance, they may outperform static load balancing variants.*

V. Numerical Results

To demonstrate the improvements of our proposed method, we provide numerical results in two different experimental settings, each using three publicly accessible and progressively larger datasets which are commonly used in ML classification experiments[1]. More precisely, the first dataset is called Pima Indians Diabetes (Diabetes), which has 768 records and 9 features and provides data to learn to classify whether a patient has diabetes or not, such that we have two possible classes for the outcome. The second dataset is named Abalone, which has 4177 records and 9 features and provides data to learn to classify the age of abalones, whose outcome is divided into 29 possible classes. The third dataset is called Predict Students' Dropout and Academic Success (Students), which has 4424 records and 37 features, and provides data to learn to classify whether students enroll, drop out or graduate from

[1]https://www.kaggle.com/datasets/uciml/pima-indians-diabetes-database, https://www.kaggle.com/datasets/hurshd0/abalone-uci, https://archive.ics.uci.edu/dataset/697/predict+students+dropout+and+academic+success

university, such that the outcome has three possible classes. In both experiments, we have used 70% of data for training and the remaining 30% for the test.

For both experiments, we have used a workstation that mounts a 2 Xeon Gold 6138 CPU and 384GB of memory. Although the 2 Xeon Gold 6138 CPU provides 40 physical cores, in all experiments we have always requested a power-of-two number of SM threads, up to 32, each bound to one physical core. This is done with a view to optimizing the workload balance across the threads, as the parallel algorithms described in Section IV mostly employ the divide-and-conquer paradigm. Every reported numerical result in the following sections is computed as the mean of 10 MC runs, each using a different seed for the random number generators. We note that 10 MC runs have shown empirically to be sufficient to extrapolate accurate average results, which is unsurprising as in this paper we have considered parallelization strategies that achieve static load balancing.

A. Performance for Fixed Problem Size

In this first experiment, we want to compare SMC, Single-Chain MCMC, and Multi-Chain MCMC in terms of run-time and accuracy for the same problem size. More precisely, this experiment has been set up as follows. Firstly, the number of samples in the SMC sampler, N, has to be an integer multiple of P, for obvious reasons. Since we are using power-of-two numbers for P, it is also convenient to do the same for N. More precisely, we use $N = \{256, 512, 1024\}$ samples. We have run the SMC sampler for $K = 10$ iterations for all datasets, whose value has been empirically set up to provide a competitive classification accuracy. Since we want to use the same problem size for all methods, we make Single-Chain MCMC draw $N \times K = N_T$ samples, and we make Multi-Chain MCMC generate N chains of K samples each. Hence, all methods are run for the same problem size, $N_T = N \times K$.

The results for this experiment are found in Table I and Figures 3, 4, 5, and 6. Table I provides the classification accuracy in all datasets for the largest problem size we have considered, i.e., $N_T = 10240$. As we can see, the SMC sampler provides roughly the same accuracy as Single-Chain MCMC, which draws a long chain of samples. However, while Single-Chain MCMC is notoriously an inherently sequential method, the SMC sampler also provides good scalability (up to a 16-fold speed-up), which, as expected from theory, improves when either the problem size increases or the dataset is bigger and, hence, more computationally intensive (see Figure 4). Therefore, by running up to $P = 32$ parallel threads, the SMC sampler is able to provide the same accuracy as Single-Chain MCMC faster by up to a 13 time factor (in Diabetes), by up to a 42 time factor (in Abalone), and by up to a 51 time factor (in Students). These results are illustrated in Figure 6, which shows the speed-up gain (vs the number of SM threads, P) of SMC in comparison with Single-Chain MCMC for the same problem size $N_T = 10240$, for which, as we said above, both methods achieve approximately equal classification accuracy.

Although Single-Chain MCMC cannot be parallelized, it is possible to run multiple MCMC chains in an embarrassingly parallel fashion. Indeed, Multi-Chain MCMC scales well with P, as shown in Figure 5. Also, this method is faster than SMC roughly by up to a factor of two (in Abalone and Students) and by up to a factor of four (in Diabetes), as shown in Figure 3. This is unsurprising due to two reasons. First, Algorithm 2 is embarrassingly parallel, while our approach is optimally bound to $O(\log_2 N)$. Second, the constant time per sample in Multi-Chain MCMC is solely due to Equations (1), (2), and (3), while updating each sample in the SMC sampler requires (apart from also having to propose and weight the new sample) extra computation given by Equations (6), (7), (8), and Algorithm 5, whose constant time is $O(\overline{M})$ (see Theorem IV.1). However, despite its competitive scalability and run-time, Multi-Chain MCMC provides worse accuracy than SMC and Single-Chain MCMC, when the problem size is fixed (see Table I). Indeed, SMC achieves good accuracy since the N samples are weighted but also updated and corrected K times through sampling and resampling. Single-Chain MCMC generates a long chain of unweighted samples, which, most certainly, have converged. Multi-Chain MCMC creates N short independent chains of $K \ll N$ unweighted samples, which, most likely, have not converged yet.

Given the run-time-vs-accuracy trade-off that Multi-Chain MCMC requires, the reader may wonder whether Multi-Chain MCMC would provide better accuracy than SMC if K is increased until the two methods run for the same elapsed time. The next section provides an answer to this question.

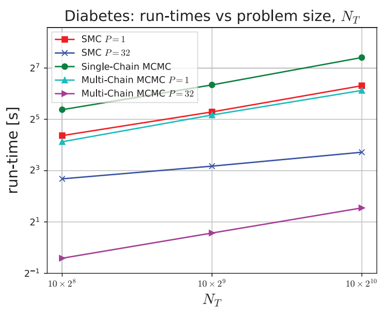
TABLE I: Classification accuracy for the same problem size $N_T = 10240$ samples, generated by each method as follows: $N = 1024$ samples updated $K = 10$ times for SMC; one chain of $N_T = 10240$ samples for Single-Chain MCMC; $N = 1024$ chains of $K = 10$ samples each for Multi-Chain MCMC.

Method	P	**Diabetes**	**Abalone**	**Students**
SMC	1	73.27%	22.48%	71.48%
SMC	32	73.27%	22.48%	71.48%
Single-Chain	1	73.78%	22.53%	71.64%
Multi-Chain	1	66.76%	20.27%	55.53%
Multi-Chain	32	66.76%	20.27%	55.53%

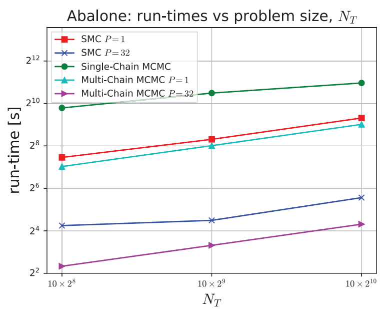
B. Accuracy for Fixed Elapsed Time

As anticipated in the previous section, in this second experiment we want to run SMC and Multi-Chain MCMC for the same elapsed time with a view to comparing their classification accuracy. To achieve this, we first consider the best run-times for both SMC and Multi-Chain MCMC for $N = 1024$, and then we increase K for Multi-Chain MCMC until the run-times of both methods match (roughly). The results are provided in Table II for Diabetes, in Table III for Abalone, and in Table IV for Students.

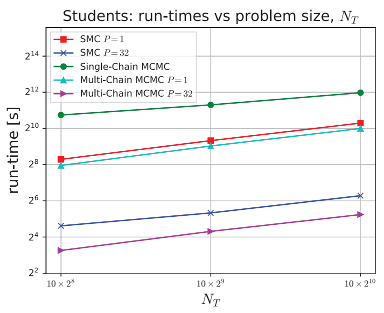
As we can see, we have had to increase K in Multi-Chain MCMC by a factor of 3.3 in Diabetes, by a factor of 1.8 in Abalone, and by a factor of 1.5 in Students with respect to the



(a) Diabetes

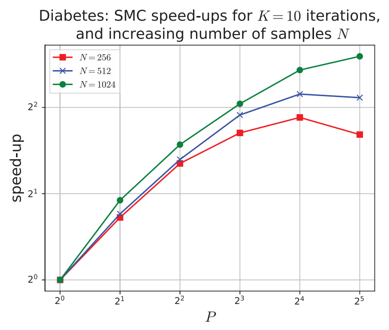


(b) Abalone

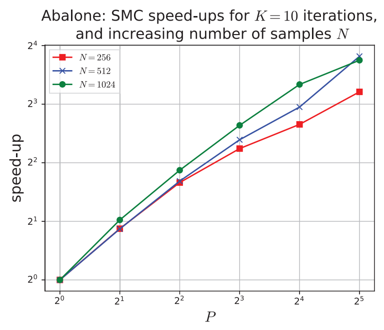


(c) Students

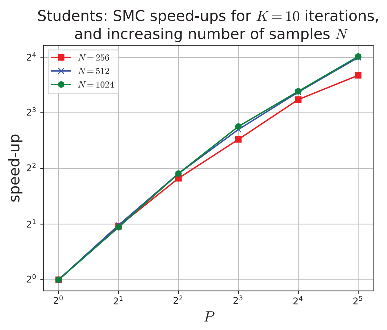
Fig. 3: Run-times for MCMC and SMC vs total sample size N_T.



(a) Diabetes

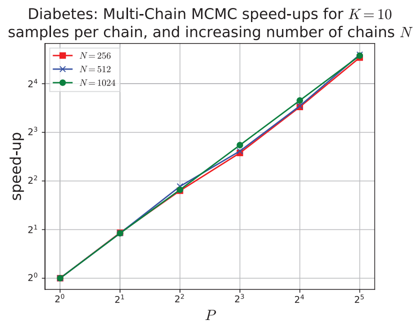


(b) Abalone

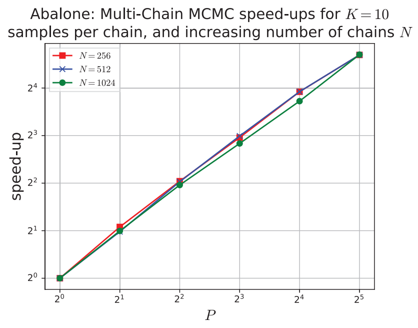


(c) Students

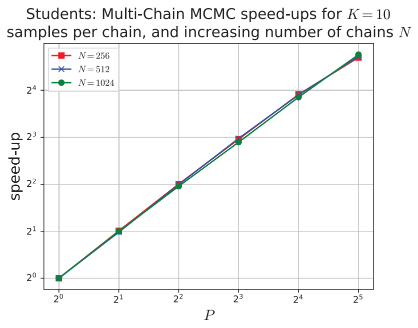
Fig. 4: Speed-ups for SMC for $K = 10$ iterations and increasing number of samples N.



(a) Diabetes



(b) Abalone



(c) Students

Fig. 5: Speed-ups for Multi-Chain MCMC for $K = 10$ samples per chain and increasing number of chains N.

previous section (in which we set $K = 10$). In all datasets, this has resulted in an improved accuracy compared to the results in Table I. More precisely, the accuracy has increased from 66.76% to 69.49% in Diabetes, from 20.27% to 20.62% in Abalone, and from 55.53% to 60.32% in Students. However, these figures are still significantly lower than those reported for SMC.

In other words, these results indicate that a parallel SMC sampler is able to compensate for the larger constant time factor than the one in parallel Multi-Chain MCMC by achieving a better accuracy per time unit, and therefore, providing a more convenient run-time-vs-accuracy compromise.

TABLE II: Accuracy per time unit - Diabetes. The * is to emphasize that the run-time is rounded to the closest integer, as running two different methods for exactly the same run-time is challenging.

Method	P	N	K	**Time**	**Accuracy**
SMC	32	2^{10}	10	13[s]*	73.27%
Multi-Chain MCMC	32	2^{10}	33	13[s]*	69.49%

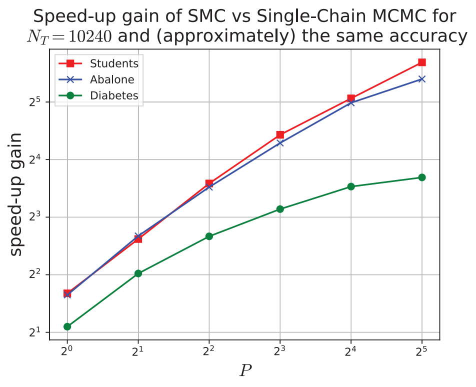


Fig. 6: Speed-up gain of SMC vs Single-Chain MCMC for (approximately) the same accuracy.

TABLE III: Accuracy per time unit - Abalone. The * is to emphasize that the run-time is rounded to the closest integer, as running two different methods for exactly the same run-time is challenging.

Method	P	N	K	**Time**	**Accuracy**
SMC	32	2^{10}	10	47[s]*	22.48%
Multi-Chain MCMC	32	2^{10}	18	47[s]*	20.62%

TABLE IV: Accuracy per time unit - Students. The * is to emphasize that the run-time is rounded to the closest integer, as running two different methods for exactly the same run-time is challenging.

Method	P	N	K	**Time**	**Accuracy**
SMC	32	2^{10}	10	77[s]*	71.48%
Multi-Chain MCMC	32	2^{10}	15	77[s]*	60.32%

VI. Conclusion

In this study, we have proposed a parallel shared memory SMC sampler for Bayesian DTs, which also includes a parallelization of the bottleneck, redistribution, which achieves optimal time complexity $O(\log_2 N)$, and takes into account that the random samples, i.e., the DTs here, have variable sizes. Hence, given that the focus is on DTs, the specific ML domain of this study is classification. By taking advantage of multi-core architectures, our approach provides up to a 16-fold speed-up for 32 parallel threads and significantly improves the performance of alternative approaches to Bayesian DTs, such as MCMC. More precisely, on three exemplar datasets for classification, our approach can provide roughly the same classification accuracy up to 51× faster for the same problem size or significantly better accuracy for the same elapsed time.

Although the results are encouraging, there is still a wide room for future improvements, both in accuracy and run-time. For example, the accuracy could be improved by using a better proposal distribution for ADTs, such as HINTS [40]. However, improving the proposal typically increases the run-time as a side effect, which can be compensated by any combination of the following ideas. First, in this manuscript, we have solely focused on SMP for CPUs. However, one could first exploit the acceleration that GPUs typically provide over CPUs. Indeed, we expect a GPU to perform significantly better on embarrassingly parallel tasks such as the sampling step in the SMC sampler. Another typical approach in High-Performance Computing is to increase the degree of parallelism by using hybrid MPI+X programming models, which fully exploit the computational power of modern supercomputers both in terms of distributed and shared memory. Another exploration avenue is to exploit alternative parallel solutions for the tasks in the SMC sampler. Indeed, in this manuscript, we have opted for static load-balancing solutions; however, given the variable-size nature of DTs, one could investigate dynamic load-balancing alternative solutions to achieve better runtime and scalability.

References

[1] A. T. Azar and S. M. El-Metwally, "Decision tree classifiers for automated medical diagnosis," *Neural Computing and Applications*, vol. 23, pp. 2387–2403, 2013.

[2] E. S. Sankari and D. Manimegalai, "Predicting membrane protein types using various decision tree classifiers based on various modes of general pseaac for imbalanced datasets," *Journal of theoretical biology*, vol. 435, pp. 208–217, 2017.

[3] A. Gajewicz, T. Puzyn, K. Odziomek, P. Urbaszek, A. Haase, C. Riebeling, A. Luch, M. A. Irfan, R. Landsiedel, M. van der Zande, *et al.*, "Decision tree models to classify nanomaterials according to the df4nanogrouping scheme," *Nanotoxicology*, vol. 12, no. 1, pp. 1–17, 2018.

[4] P. Kazemi, A. Ghisi, and S. Mariani, "Classification of the structural behavior of tall buildings with a diagrid structure: A machine learning-based approach," *Algorithms*, vol. 15, no. 10, pp. 349–372, 2022.

[5] E. Drousiotis, L. Shi, and S. Maskell, "Early predictor for student success based on behavioural and demographical indicators," in *International Conference on Intelligent Tutoring Systems*, pp. 161–172, Springer, 2021.

[6] V. Schetinin, L. Jakaite, and W. Krzanowski, "Bayesian averaging over decision tree models for trauma severity scoring," *Artificial Intelligence in Medicine*, vol. 84, pp. 139–145, 2018.

[7] G. I. Valderrama-Bahamóndez and H. Fröhlich, "Mcmc techniques for parameter estimation of ode based models in systems biology," *Frontiers in Applied Mathematics and Statistics*, vol. 5, no. 55, 2019.

[8] B. Hernández, S. R. Pennington, and A. C. Parnell, "Bayesian methods for proteomic biomarker development," *EuPA Open Proteomics*, vol. 9, pp. 54–64, 2015.

[9] J. L. Bez, F. Z. Boito, R. Nou, A. Miranda, T. Cortes, and P. O. A. Navaux, "Detecting i/o access patterns of hpc workloads at runtime," in *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 80–87, 2019.

[10] A. R. Linero, "Bayesian regression trees for high-dimensional prediction and variable selection," *Journal of the American Statistical Association*, vol. 113, no. 522, pp. 626–636, 2018.

[11] H. A. Chipman, E. I. George, and R. E. McCulloch, "Bayesian cart model search," *Journal of the American Statistical Association*, vol. 93, no. 443, pp. 935–948, 1998.

[12] D. G. Denison, B. K. Mallick, and A. F. Smith, "A bayesian cart algorithm," *Biometrika*, vol. 85, no. 2, pp. 363–377, 1998.

[13] V. Schetinin, J. E. Fieldsend, D. Partridge, W. J. Krzanowski, R. M. Everson, T. C. Bailey, and A. Hernandez, "The bayesian decision tree technique with a sweeping strategy," *arXiv preprint cs/0504042*, 2005.

[14] D. G. Denison, C. C. Holmes, B. K. Mallick, and A. F. Smith, *Bayesian methods for nonlinear classification and regression*, vol. 386. John Wiley & Sons, 2002.

[15] W. Hu, R. A. O'Leary, K. Mengersen, and S. Low Choy, "Bayesian classification and regression trees for predicting incidence of cryptosporidiosis," *PLOS ONE*, vol. 6, pp. 1–8, 08 2011.

[16] A. S. Malehi and M. Jahangiri, “Classic and bayesian tree-based methods,” in *Enhanced Expert Systems* (P. Vizureanu, ed.), ch. 3, pp. 27–51, Rijeka: IntechOpen, 2019.

[17] J. M. Byrd, S. A. Jarvis, and A. H. Bhalerao, “Reducing the run-time of mcmc programs by multithreading on smp architectures,” in *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–8, IEEE, 2008.

[18] J. Ye, A. M. Wallace, A. Al Zain, and J. Thompson, “Parallel bayesian inference of range and reflectance from ladar profiles,” *Journal of Parallel and Distributed Computing*, vol. 73, no. 4, pp. 383–399, 2013.

[19] E. Drousiotis and P. G. Spirakis, “Single mcmc chain parallelisation on decision trees,” in *Learning and Intelligent Optimization: 16th International Conference, LION 16, Milos Island, Greece, June 5–10, 2022, Revised Selected Papers*, pp. 191–204, Springer, 2023.

[20] R. Mohammadi, M. Pratola, and M. Kaptein, “Continuous-time birth-death mcmc for bayesian regression tree models,” *J. Mach. Learn. Res.*, vol. 21, pp. 1–26, jan 2020.

[21] M. T. Pratola, H. A. Chipman, J. R. Gattiker, D. M. Higdon, R. McCulloch, and W. N. Rust, “Parallel bayesian additive regression trees,” *Journal of Computational and Graphical Statistics*, vol. 23, no. 3, pp. 830–852, 2014.

[22] D. Francom, B. Sansó, A. Kupresanin, and G. Johannesson, “Sensitivity analysis and emulation for functional data using bayesian adaptive splines,” *Statistica Sinica*, vol. 28, no. 2, pp. 791–816, 2018.

[23] 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.

[24] 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, pp. 342–362, 2021.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] A. Chatzopoulou, Á. F. García-Fernández, E. Pyzer-Knapp, and S. Maskell, “Smc samplers for bayesian optimisation and discovery of additive kernel structure,” in *2021 IEEE 24th International Conference on Information Fusion (FUSION)*, pp. 1–8, 2021.

[30] E. Drousiotis, P. G. Spirakis, and S. Maskell, “Parallel approaches to accelerate bayesian decision trees,” *arXiv preprint arXiv:2301.09090*, 2023.

[31] 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.

[32] 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.

[33] T. Hastie, R. Tibshirani, J. H. Friedman, and J. H. Friedman, *The elements of statistical learning: data mining, inference, and prediction*, vol. 2. Springer, 2009.

[34] T. Li, M. Bolic, and P. M. Djuric, “Resampling methods for particle filtering: Classification, implementation, and strategies,” *IEEE Signal Processing Magazine*, vol. 32, no. 3, pp. 70–86, 2015.

[35] M. T. Pratola, “Efficient Metropolis–Hastings Proposal Mechanisms for Bayesian Regression Tree Models,” *Bayesian Analysis*, vol. 11, no. 3, pp. 885 – 911, 2016.

[36] Y. Wu, H. Tjelmeland, and M. West, “Bayesian cart: Prior specification and posterior simulation,” *Journal of Computational and Graphical Statistics*, vol. 16, no. 1, pp. 44–66, 2007.

[37] A. P. Diéguez, M. Amor, and R. Doallo, “Efficient scan operator methods on a gpu,” in *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, pp. 190–197, 2014.

[38] 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), p. 1–11, Association for Computing Machinery, 2021.

[39] 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.

[40] M. Strens, “Efficient hierarchical mcmc for policy search,” in *Proceedings of the Twenty-First International Conference on Machine Learning*, ICML ’04, (New York, NY, USA), pp. 97–104, Association for Computing Machinery, 2004.