P-STreeD

A Multithreaded Approach for DP Optimal Decision Trees

Albert-Alexandru Sandu[1]
Supervisor(s): Emir Demirović[1]**, Jacobus G. M. van der Linden**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 23, 2024

Name of the student: Albert-Alexandru Sandu
Final project course: CSE3000 Research Project
Thesis committee: Emir Demirović, Jacobus G. M. van der Linden, D.M.J. Tax An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

Decision trees are valued for their ability to logically and transparently classify data. While heuristic methods to compute such trees are efficient, they often compromise on accuracy, prompting interest in Optimal Decision Trees (ODTs), which have the best misclassification score for a given tree size limit and training dataset. The dynamic programming (DP) approach for ODTs has shown improvements over alternatives such as mixed-integer or constraint programming. That being said, it requires further improvements to handle exponential runtime scaling with the depth of the tree and the number of features of the dataset. Leveraging modern hardware, such as multiple CPU cores, offers a promising solution to improve efficiency. This paper proposes a multithreading method for DP ODTs which we apply to STreeD specifically. We introduce a shared memory model and determine which components of the original program can be made local to the threads. We investigate whether it is more efficient to start multithreading at the root of the search tree than near its leaf nodes. We find the former to be superior, resulting in faster runtimes and less shared resource access. Empirical evaluations against the state of the art demonstrate better runtimes, particularly beneficial for large datasets. Finally, thread scaling analyses reveal substantial speedups, exceeding 2.5 times with four threads, highlighting our approach's effectiveness for computationally-intensive tasks.

1 Introduction

Decision trees are data structures which allow us to logically and transparently classify data. We can divide the data into smaller parts based on various features of the objects it encompasses. Such trees have applications in Machine Learning and are popular thanks to their simplicity, interpretability, and ability to handle both numerical and categorical data (Costa & Pedreira, 2023). Moreover, they can capture non-linear relationships in data.

We understand an Optimal Decision Tree (ODT) as the best tree in terms of misclassifications for a given depth limit. Unlike traditional greedy decision tree heuristics, such as CART (Breiman et al., 1984) and C4.5 (Quinlan, 1993), which may stop growing the tree too early or create suboptimal splits, optimal decision tree algorithms choose the best feature and threshold for splitting the data at each step. This approach often results in trees with better generalization performance and interpretability when compared to heuristic-based methods (Aglin et al., 2020).

However, as proven by Hyafil and Rivest (1976), finding the optimal tree is a NP-Hard problem. For large datasets, scalability is an issue, impacted by the number of features and the depth of the tree. Some other limitations of ODTs are that their optimality is only guaranteed for the training data (thus, they could be prone to overfitting) and that (for many methods) they assume already binarized data; the binarization process may affect their performance.

There are various approaches towards constructing optimal decision trees, such as mixed-integer programming (MIP), constraint programming (CP), and SAT based ones. Dynamic programming (DP) based solutions were shown to usually perform better than their counterparts (Aglin et al., 2020). That being said, improvements to DP approaches are still needed because constructing the tree scales exponentially with depth and number of binary features (Demirović et al., 2022). This makes tuning and training for complex data time-consuming and inaccessible, depending on one's resources. Improving scalability is important because the amount of data we work with is ever-increasing, as technologies such as IoT further

develop and encompass multiple complex systems (Costa & Pedreira, 2023). Examples of applicability within IoT for decision trees are in security (Puthal et al., 2022).

One of the possible improvements to DP based approaches is multithreading. The main motivation behind this is that DP divides a problem into multiple, independent subproblems, which we can then use to find the solution for the original one. In the case of constructing ODTs, the left and right subtree are independent subproblems. This means that computing either of them should not depend on the other. Therefore, threads could be leveraged for the two subtrees to be done in parallel. In reality, the subtrees are not truly independent due to various optimizations in the state of the art. This means that multithreading for this case is not trivial, but there are still parts of the DP approach to analyze and potentially multithread.

Our contribution We investigate the feasibility of applying multi-threading to DP-based ODTs. The optimality of the tree must be maintained (so the misclassification score should be identical to that of a sequential algorithm, despite the possibility of a different tree output). We identify the most promising sections of a DP approach to parallelize and outline a shared memory model to implement the multithreading. We implement the most promising approach, which parallelizes the feature split loop. This implementation is done over STreeD (van der Linden, de Weerdt, & Demirović, 2023), using C++ threads.

We investigate whether applying multithreading is more efficient close to the root of the search tree or to its leaf nodes. Our experimental results show that the root-based strategy results in better runtimes and less cache access.

We compare the new program (which we call P-STreeD) with the state of the art: STreeD, Murtree, DL8.5. Our method is generally faster than the others, with the difference being most pronounced for larger datasets. We measure the speed-up achieved through multi-threading and how it relates to the number of threads, but also the dimensions of the problem (tree depth and dataset size). We find speed-ups of more than 2.5 times for some datasets.

The following describes the structure of the paper. Section 2 outlines the existing work in the literature and how it relates to our contribution. Section 3 gives the necessary background to understand our method through a review of Murtree. Section 4 contains our main contribution. It explains how we parallelized STreeD and what changes had to be made to the original algorithm. Section 5 features our experimental set-up and results. Section 6 is a reflection on the reproducibility and ethical aspects of our research. Section 7 provides a discussion on the results of the experiments. Italso features the applicability of our contribution towards different DP ODT algorithms, as well as how it could be adapted to work on a supercomputer. Section 8 summarizes the main findings of this paper, our results, as well as recommendations for future work.

2 Related Work

The field of decision tree learning has seen significant advancements in recent years, with researchers exploring various optimization approaches to improve scalability. Traditional heuristic methods like CART (Breiman et al., 1984) and C4.5 (Quinlan, 1993) have long been favored for their effectiveness and scalability. However, with increasing computational power, optimal decision tree search for limited depth has become feasible, prompting researchers to explore MIP, SAT, CP, and DP-based approaches.

MIP-based approaches, such as those proposed by Bertsimas and Dunn (2017) and Verwer and Zhang (2019), offer the advantage of finding globally optimal trees but suf-

fer from scalability issues, particularly for medium to large datasets. Nevertheless, they have shown promise in optimizing various objectives, including misclassification error and fairness metrics. Recently, Hu, Rudin, and Seltzer (2019) introduced an algorithm that balances misclassifications and the number of nodes, achieving good performance with sufficient sparsity.

SAT-based approaches, pioneered by Narodytska et al. (2018), aim to construct the smallest tree that perfectly describes the dataset, focusing on zero misclassifications, although there are variations such as MAX-SAT that maximize accuracy instead.

DP-based approaches started seeing more improvements recently. Examples include DL8 (Fromont & Nijssen, 2007) and DL8.5 (Aglin et al., 2020), which build upon the work of Agrawal et al. (1996), in the itemset mining literature. These provide better scalability compared to MIP and SAT solvers. DL8.5 introduces upper and lower bounding for nodes, allowing for more efficient pruning and a considerably faster algorithm. Lin et al. (2020) introduced several advancements with GOSDT. They propose a generalized approach to optimize sparse decision trees, accommodating various objective functions beyond accuracy. They allow continuous features and present a new search space representation. The algorithm works best with datasets that have a small to moderate number of features. They also compute bounds asynchronously, using a priority queue to schedule bound updates to the parents from the children. This was of particular interest to us when devising our method. That being said, there are conceptual differences between GOSDT and STreeD and we decided to implement our method for STreeD. Using a similar strategy to that of theirs would have involved abstracting STreeD into a system of tasks and making it iterative instead of recursive. It would have also involved using a dependency graph in order to find which subproblem to schedule.

Demirović et al. (2022) introduce MurTree, which allows for constraints on tree depth and node count while employing dynamic programming and search principles. It introduces a specialized algorithm for depth-two trees, leveraging frequency counting due to the binary nature of input data. Additionally, they propose a similarity-based bounding mechanism, facilitating efficient pruning of the search space by examining previously computed subtrees. Extensions to DL 8.5 are also presented, including enhancements to caching techniques to consider depth and node constraints. Furthermore, they provide novel implementations of two caching schemes and discuss dynamic node exploration strategies. van der Linden et al. (2023) introduced a generalized framework for optimizing separable objectives and constraints, based on the Murtree algorithm: STreeD. They establish necessary and sufficient conditions for separability, extending the applicability of dynamic programming approaches. STreeD also outperforms general-purpose solvers in scalability while maintaining similar or better accuracy and size optimization.

In terms of multi-threading, efforts have been made to parallelize dynamic programming techniques in general. Stivala et al. (2010) devised a general technique for parallelizing top-down dynamic programming approaches through the use of lock-free hash-tables for memoization, as well as the randomization of subproblem computation. They explained how this leads to a higher probability of thread computations to diverge, thus scheduling less sub-problems to be solved by the same thread simultaneously. These contributions result in substantial speed-ups over their serial counter-parts. That being said, the approach requires careful, problem-specific, analysis. We attempted to build upon these ideas but haven't found success, as explained in Section 4.5.

Narlikar (1998) has leveraged divide-and-conquer parallelism for the original C4.5 decision tree building algorithm. They exploited this at the outer level (throughout the nodes),

and at the inner level (within each node) by improving sorting operations. The approach is adaptable to varying numbers of CPU cores and introduces a space-efficient scheduling algorithm. We were inspired by their contributions to investigate the terminal solver parallelization approach featured in Section 4.2.

3 Preliminaries

Our approach is based on STreeD (van der Linden et al., 2023), which in turn builds on many ideas introduced in Murtree (Demirović et al., 2022). We consider STreeD to be representative of the class of DP ODT algorithms, especially because of its ability to generalize tasks. By using it as our base, we can in turn extend our contributions to other similar DP ODT solutions. This section represents a review of the Murtree algorithm, along with optimizations relevant to our use case.

3.1 Murtree

Dynamic Programming Formulation The formulation in Equation 1, as given by Demirović et al. (2022), describes the high-level ODT computation approach of Murtree. The input parameters are the dataset ($\mathcal{D}$), its features ($\mathcal{F}$), as well as the upper bounds on the depth of the tree (d) and the number of feature nodes (n). The first two cases deal with depth and feature node limits. The third case represents nodes where the label is determined by the majority class, whereas the fourth represents the general recursion case. It involves iterating over the possible feature splits and feature node count distributions for the left and right subtrees in order to find the split with the minimum misclassification score.

$$T(\mathcal{D}, d, n) = \begin{cases} T(\mathcal{D}, d, 2^d - 1) & \text{if } n > 2^d - 1 \\ T(\mathcal{D}, n, n) & \text{if } d > n \\ \min\{|\mathcal{D}^+|, |\mathcal{D}^-|\} & \text{if } n = 0 \vee d = 0 \\ \min\{T(\mathcal{D}(\bar{f}), d-1, n-i-1) & \text{if } n > 0 \wedge d > 0 \\ \quad +T(\mathcal{D}(f), d-1, i) : f \in \mathcal{F}, i \in [0, n-1]\} & \end{cases} \tag{1}$$

Similarity-Based Lower Bounding Murtree introduces a method for deriving a lower bound on the misclassification score of an optimal decision tree. This technique compares a new dataset D_{new} to a previously analyzed dataset D_{old}. The lower bound is defined as:

$$\text{LB}(D_{\text{new}}, D_{\text{old}}, d, n) = T(D_{\text{old}}, d, n) - |D_{\text{out}}|$$

where D_{out} is the set of instances in D_{old} but not in D_{new}. The algorithm maintains two datasets per depth, using the most similar ones to compute the lower bound, and caches optimal solutions and bounds for efficient reuse.

Subtree computation order The algorithm first computes the left subtree. If it is infeasible, it skips the computation of the right one. It then subtracts the misclassification of the left subtree from the upper bound. By doing so, it establishes a new upper bound for the right subtree before computing it recursively.

Branch caching Subtrees can be cached as branches, defined as sets of features extending from the root to any given node. Each branch is stored as a sorted array of integers, where features are encoded based on their index and whether they are present or not. A standard hash function is utilized to compute the hash value of these arrays. It is worth

noting that different branches may correspond to the same dataset, which is not detected, potentially leading to the recomputation of equivalent subproblems.

Dataset caching Subtrees can also be represented more generally by utilizing datasets. At the beginning of the algorithm, each instance is assigned a unique identifier, with instances within each class sorted according to these identifiers. The hash value of a dataset is computed using these instance identifiers and stored for subsequent use, enabling the algorithm to determine the equivalence of two datasets in linear time. Dataset caching is capable of correctly identifying all equivalent subproblems.

Node selection strategy Murtree uses dynamic ordering in order to determine the first subtree to explore. It computes a heuristic on the number of misclassifications for both children and chooses the one with the higher value. This allows the algorithm to prune infeasible solutions earlier. STreeD opts for a different approach though. It chooses the first child node to explore based on which one has the larger data size.

Terminal solver Murtree uses a specialized algorithm for computing trees of depth-2. It has two phases: frequency counting and the optimal tree computation. The first phase involves computing for each feature pair the number of instances where both are present. This way, the second phase avoids directly manipulating the data itself. Said phase has a better time complexity than the generalized ODT case due to manipulating tree properties. The algorithm also re-uses past computations; instead of computing the frequency counts each time the terminal solver is called, it maintains the previous state and updates it instead. This is based on the following observation: datasets differing by small amounts have similar frequency counts. It uses set operations, which are efficient in the context of datasets. Both Murtree and STreeD use two such terminal solvers. The terminal solver with the most similar dataset to that of the current subproblem is chosen. Around 90% of STreeD's runtime resides in the terminal solver procedure.

4 Multithreading for STreeD

To the end of parallelizing STreeD, we have outlined the shared memory of the algorithm, as well as certain considerations for the correctness of a multithreaded approach. It is worth noting that our contribution is based on the full STreeD algorithm. Therefore, the effect of various features of STreeD on the feasibility of certain multithreading designs is also investigated. We then present in detail P-STreeD: an approach parallelizing the feature loop within the fourth case of Equation 1, using a fixed amount of threads.

4.1 Shared memory

Cache The dataset and branch caches are global in STreeD through a single object. This has to be shared in our multithreaded version(s). The cache is used at many points throughout the algorithm. Efficiently using it is required to avoid computing the same subproblem.

Terminal solvers STreeD uses two terminal solvers. They keep track of previous state as well as their most recently computed solution. They need to be locked each time they are accessed. While it is possible to modify the data structure to not keep track of the solution itself (and therefore avoid data races involving it), storing the previous frequency counts still poses an issue. We risk corrupting them and removing them is not feasible due to their use as an optimization. Therefore, we choose not to modify the terminal solver algorithm or its afferent data structures. Instead of locking, it is possible to assign fixed terminal solvers to threads.

Similarity lower bound computer STreeD's updates to the cache using similarity lower bounds employ a data structure where similar solutions are stored. This data structure is not thread-safe, therefore, mutual exclusion is needed to use it correctly in a multithreaded context.

Solutions Any multithreading approach involves improving an area of the computation of the subtree. Because most of the runtime resides in the terminal solver, any approach will involve parallelizing terminal solver calls or the terminal solver itself. Either of these modify a solution. Updating it requires mutual exclusion between the threads. Whereas the cache and terminal solvers are global (so they should be accessed by any threads) the solutions updated should be shared only between threads at the same level. If the main thread of the program creates n threads to solve the subproblems, the local variable representing a solution at the point of creating the threads should be shared only with these n threads. Indifferent of the multithreading approach taken, if these threads create others at any other point, their "children" should not update this original variable.

4.2 Overview of other investigated multithreading designs

In this subsection, we outline the multithreading designs which we considered, but not used, for our implementation. Each design is briefly explained, along with reasons why it was not considered feasible or worthwhile for our use case.

Thread each call to $T(\mathcal{D}, d, n)$ This naive approach involves trying to apply multithreading to any recursive call. We find it to be infeasible. On one hand, it implies creating and managing too many threads. In the case of using a thread pool, it still implies creating too many tasks to assign to the worker threads. On the other hand, it is incompatible with computing a child node first to determine the upper bound for the other. If we were to enforce computing a child node first then the multithreaded solution would be practically equivalent to the sequential one, resulting in no parallelism.

Thread the feature loop within the terminal solver On a level of abstraction, this is the design we proceed with, but within the terminal solver. The idea behind this is that if the terminal solver represents 90% of the runtime of the algorithm, then improving it would have a strong effect. Despite this, we find this approach to not be worth pursuing. This is because most of the runtime of the terminal solver itself is spent in the frequency counter. Therefore, synchronization is needed on the frequency counter, resulting in too much contention. The shared memory model required is also completely different, involving cost calculators, the left and right solution, and the branch context.

Thread the node budget distribution loop Here we refer to the inner loop within the fourth case of Equation 1. This approach is comparable to the one we proceed with, requiring a similar shared memory model and dividing left tree sizes, instead of the features, into blocks. The datasets for the subproblems computed in parallel might have a higher degree of similarity than in the other designs, thus triggering more cache hits and retrieving solutions earlier. We find this to not be worth pursuing though. It is ineffective in speeding up the computation of complete trees (they do not require the node budget distribution). Each thread call also implies heavier overhead due to a larger context, as more variables have to be passed. Moreover, there is a larger degree of threads being created or tasks assigned to worker threads.

4.3 Parallelizing the feature loop within the general case

Our chosen design for parallelizing STreeD is applying multithreading to the outer loop featured within the general recursion case. The high-level overview can be found in Figure 1.

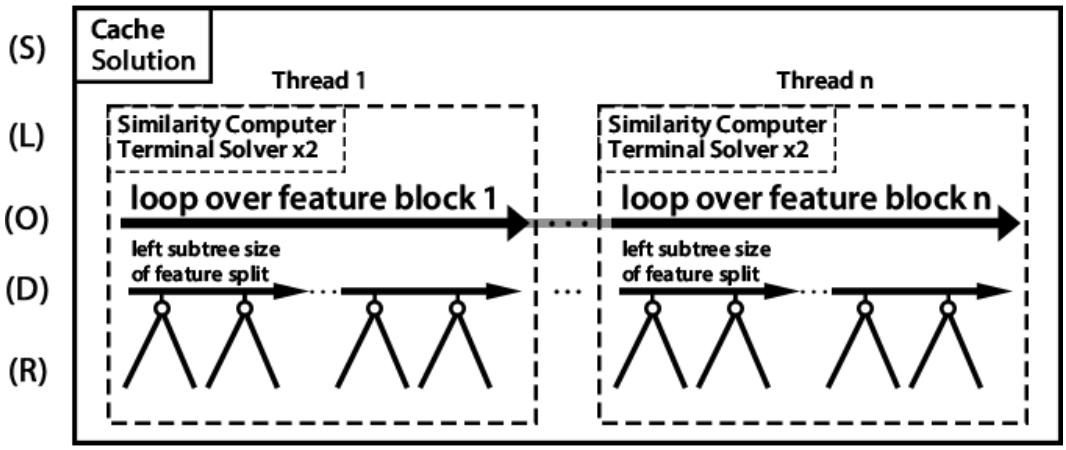

Figure 1: Overview of the design for P-STreeD. (S) denotes the global shared memory, (L) local memory of the thread which would've been shared, (O) the feature loop, (D) the node distribution budget loop for the left subtree and (R) the recursive calls for the left and right subtree.

The design approach involves splitting the features to be looped over into several blocks. The original logic of the loop is separated into a function which is given to the threads to execute. All of these threads have to finish before the master thread continues its computation. We place mutual exclusion on the shared solution the threads are working on together. This shared solution is only updated and locked within the loop itself.

4.4 Threading strategy

We use at most as many threads as physical CPU cores. We do not base the number of threads on that of logical processors. That is because our tasks are computationally heavy, we are not bound by input or other forms of waiting. Therefore, using as many threads as logical processors does not improve the runtime and poses unnecessary overhead from the system managing more threads than it can run concurrently.

In the case of ODT DP, one of the consequences of multithreading is that a larger part of the problem search space is explored than in the sequential version. Multiple threads can compute equivalent subproblems, with the same features, but in a different order. Because neither of them is finished, none of them is able to retrieve a solution from the cache. Therefore, there is redundancy, which results in overhead.

Our approach starts parallelizing at the root level. Alternatively, multithreading could be applied close to the leaf nodes, before calls to the terminal solver. This results in too many such redundant solutions. This approach also involves an exponential amount of tasks to be created, implying a degree of overhead. The hypothesis that parallelizing closer to the root is faster is tested in the next section.

4.5 Feature order strategy

We use the in-order feature order. Stivala et al. (2010) recommend using a random order for the subproblems in order to encourage search space divergence for the threads. We implemented this by randomizing the feature vector for each level of the tree. We also investigated a variant where we randomized the first level after starting the multithreading procedure. We found no improvement from either. We believe this is because the ODT DP algorithm makes use of similarity and other such optimizations benefiting from a fixed order.

4.6 Terminal solver and similarity computer strategy

Incremental solving is an important part of the original algorithm. A degree of compatibility between the dataset of the current subproblem and the terminal solver it is going to use is required. STreeD and Murtree use two terminal solvers but results in much contention over them. To address this, we use two terminal solvers for each thread. This completely negates the need of synchronization. This applies to the similarity lower bound computer too (of which there is only one in STreeD and Murtree). We use one such computer for each thread.

5 Experimental Setup and Results

5.1 Experimental setup

Method We implement P-STreeD over the STreeD C++ library source code. We use C++11 threads; task scheduling is done with std::async and std::future. We evaluate the runtime performance of constructing the tree in our approach when parallelizing at the root level and close to the leaf nodes. We compare how the better performing variation compares with the state of the art: Murtree, STreeD, and DL8.5. Our hypothesis is that P-STreeD performs better than the state of the art. Then, we investigate how the speed-up scales with the maximum depth and number of binary feature nodes. We do not compare with GOSDT as Demirović et al. (2022) already compared Murtree with it and it is much faster.

The machine used for the experiments features an Intel i5-9300H CPU, with 4 physical cores and a base speed of 2.4 GHz. It also has 8GB of RAM running at 2667 MHz. The code is compiled using GCC. We run each test 10 times.

Metrics In tables, we report the averages of the experiment runtimes, as well as other aspects such as calls to the terminal solver, cache hits and misses. We compare the runtimes of our method with STreeD and Murtree using the two-sided Wilcoxon Signed Ranked Test, which is suitable for non-normally distributed data. The results from the test are a p-value and the W-statistic. If the p-value is below 0.05, we consider the results statistically significant. The larger the W-statistic is, the greater the difference is between the two methods. To confirm our hypothesis, the obtained W-statistic must be larger than the critical-value, which is 8 (for our chosen p-value and number of runs per dataset).

5.2 Root versus leaf parallelization

We aim to determine whether parallelizing close to the root is more efficient than close to the leaf nodes. We measure the runtime, number of terminal calls, cache hits and cache misses. We run P-STreeD with both branch and dataset caching.

Our base assumption is that the leaf-based version explores a larger part of the search space due to the proximity of the multithreading procedure to the terminal solver - the base case of the recursion. Thus, a larger degree of terminal calls, cache hits and misses are expected. Table 1 features results for $d = 5, n = 31$, which represents a complete tree for the given depth. The root-based scenario outperforms both the baseline and the leaf-based scenario for all datasets.

Experiments are also done for $d = 4, n = 15$ and $d = 5, n = 24$. The latter represents a non-complete tree. We are interested in this scenario as tuning might be done for the number of feature nodes too, not only for the depth. The results can be found in Table 4 and Table 5. For the former, results are quite similar for all datasets. The leaf-based version tends to have more terminal calls and is generally slower than the root, but is close to or faster than the baseline.

To summarize, the root-based version of P-STreeD clearly outperforms the leaf-based one, having better runtimes, as well less terminal calls or interactions with the cache.

Table 1: Comparison of statistics between multithreading versions for depth $d = 5$ and maximum number of feature nodes $n = 31$. $|D|$ and $|F|$ refer to the number of instances and features of the dataset, respectively. *Type* refers to where we apply the multithreading procedure; *none* is a version of our program with multithreading disabled. Time measured in seconds. *D2S Calls* denotes calls to the terminal solver. *D2S Calls, Cache Hits* and *Cache Misses* are measured in thousands. Best runtimes marked with bold.

Dataset	$\|\mathcal{D}\|$	$\|\mathcal{F}\|$	Type	Time	D2S Calls	Cache Hits	Cache Misses
anneal	812	93	root	**<1**	9444	1469	30920
			leaf	6	13667	4059	58294
			none	2	8300	1543	27235
diabetes	768	112	root	**14**	96310	22361	215745
			leaf	28	100305	33609	294868
			none	31	95094	20985	208154
heart-cleveland	296	95	root	**3**	32733	2767	85983
			leaf	8	31699	3246	106454
			none	7	27885	2119	74101
hepatitis	137	68	root	**<1**	1427	55	3460
			leaf	<1	1470	68	3920
			none	<1	1273	42	3090
kr-vs-kp	3196	73	root	**2**	9854	584	26953
			leaf	8	9625	980	36721
			none	5	8848	528	24167
mushroom	8124	119	root	<1	1160	12	3143
			leaf	**<1**	383	2	1028
			none	<1	232	<1	603
pendigits	7494	216	root	**119**	56137	1345	148188
			leaf	208	63778	1728	177897
			none	227	47315	1036	124998

5.3 State of the art comparison

We compare the runtimes of DL8.5, STreeD, P-STreeD (parallelizing at the root) and Murtree. We run Murtree, STreeD and P-STreeD with dataset caching enabled and branch caching disabled for a fair comparison. We note that we cannot set the maximum number of feature nodes for DL8.5. Therefore, we only set the depth. The results for $depth = 5, n = 31$ can be found in Table 2. For any tests taking more than 1 second, P-STreeD outperforms the state of the art. The only exceptions are mushroom ($depth = 5, n = 24$), ionosphere ($depth = 5, n = 31$), and splice-1 ($depth = 5, n = 24$). For tests with runtimes below 1 second, P-STreeD is generally slower than the other methods (including STreeD). The results for $depth = 4, n = 15$ and $depth = 5, n = 24$ can be found in Table 6 and Table 7, respectively. The results for these are similar, with P-STreeD being faster for longer tests and exhibiting the same behavior for the specific datasets mentioned earlier.

Table 2: Comparison of runtime performance between algorithms for depth $d = 5$ and maximum number of feature nodes $n = 31$. $|D|$ and $|F|$ refer to the number of instances and features of the dataset, respectively. Timeouts of 600 seconds represented with $--$. Best runtimes marked with bold.

Dataset	$\|\mathcal{D}\|$	$\|\mathcal{F}\|$	DL8.5	STreeD	P-STreeD	Murtree
anneal	812	93	5	2	**<1**	2
audiology	216	148	9	<1	<1	**<1**
breast-wisconsin	683	120	12	3	**<1**	2
diabetes	768	112	72	35	**14**	20
fico-binary	10459	17	<1	2	<1	2
german-credit	1000	112	114	102	**42**	55
heart-cleveland	296	95	15	7	**3**	4
hepatitis	137	68	2	<1	<1	**<1**
ionosphere	351	445	--	301	370	**135**
kr-vs-kp	3196	73	12	6	**2**	7
lymph	148	68	2	<1	<1	**<1**
mushroom	8124	119	18	<1	<1	**<1**
pendigits	7494	216	--	271	**119**	235
tic-tac-toe	958	27	<1	<1	**<1**	<1
vehicle	846	252	258	146	**77**	137
yeast	1484	89	43	12	**6**	14
Average rank			3.72	2.72	**1.55**	2.00

We compare our method with STreeD and Murtree (as DL8.5 is generally outperformed by both of them) using the Wilcoxon Signed-Ranked Test. The results can be found in Table 3. For $depth = 5, n = 24$ against Murtree, the p-value is too large (above 0.05), indicating that our test is not statistically significant. For $depth = 4, n = 15$ against STreeD, the recorded W-statistic (1.0) is lower than the critical value of 8. This indicates that our hypothesis that P-STreeD is faster than STreeD in this case is rejected. indicating that our hypothesis is rejected. For the other cases, it is confirmed that our method is faster than STreeD and Murtree.

Table 3: Wilcoxon Signed-Rank Test results for depths $d = \{4, 5\}$. Each entry includes the W-Statistic, followed by the p-value in parentheses. Significant results marked with bold.

depth	n	P-STreeD vs Murtree	P-STreeD vs STreeD
4	15	**14.0 (0.0008)**	1.0 (0.0000)
5	24	63.0 (0.3465)	**26.0 (0.0077)**
5	31	**35.0 (0.0268)**	**30.0 (0.0139)**

5.4 Speed-up scaling

We compare the speed-ups for different numbers of threads used. The datasets we test on are diversified in terms of their number of instances and features. The results can be found in Figure 2. P-STreeD generally gets faster with more threads, although some smaller datasets experience slow-downs instead.

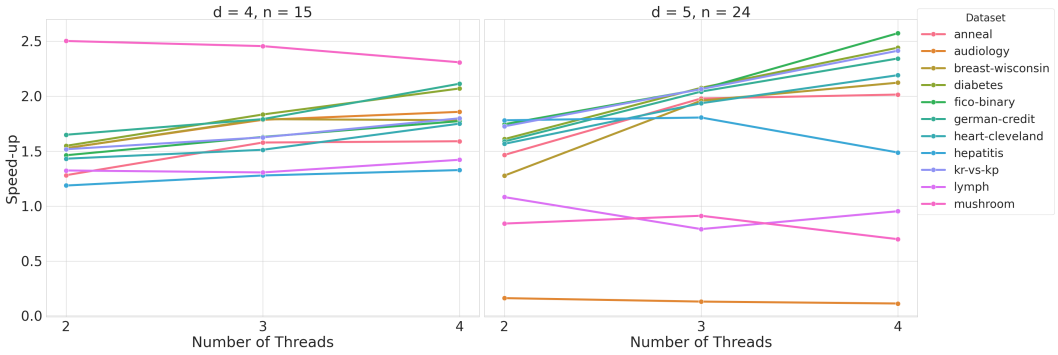

Figure 2: Speed-up comparisons by number of threads {2-4}. Depth and maximum number of feature nodes are $d = 4, n = 15$ and $d = 5, n = 24$. Baseline is our method with a single thread.

6 Responsible Research

6.1 Reproducibility

Our approach is implemented on top of STreeD. If one were to follow the design outlined they would have similar findings. The runtimes for P-STreeD and its variations might differ, depending on system configuration. The machine employed for the experiments uses Windows 11. Running on a different operating system might have different results due to kernel-level machinations and thread scheduling mechanisms such as pre-emption (which is entirely out of our control). The code for our experimental set-up and the full results can be found at `github.com/albsd/P-STreeD-experiments`. The code for our implementation is in a branch of a private repository which will be made public in the future. The datasets can be found in the Murtree paper (Demirović et al., 2022).

6.2 Ethical considerations

Our research in itself doesn't pose any ethical considerations. Decision trees can be used for any sort of task. Training them for particular domains such as healthcare might require the use of sensitive data. Applying them in various scenarios might have different ethical implications but it is impossible to list all of those. Within our research, we used publicly available datasets complying with ethical and legal standards.

7 Discussion

We discuss the results of the experiments conducted in Section 5, as well as some aspects of the applicability of our contribution. We address the performance of the leaf-based version of P-STreeD. Then, we explain the best use cases for our program. Finally, we iterate how our approach can be applied to other similar DP ODT methods.

7.1 Leaf-based parallelization

Parallelizing close to the root of the search tree was shown to be more efficient than close to the leaf nodes. The leaf-based version performs similarly to the baseline for complete trees. The differences are much more pronounced for the non-complete tree described by $depth = 5, n = 24$, although the difference in terminal solver calls is not as noticeable. There are instances when the leaf-based version has less terminal calls than the root-based one, but it still has more cache hits and misses. An example is for *diabetes*, for $depth = 4, n = 15$. A possible explanation we could offer for this behavior is that terminal calls may be done in parallel, with bounds and solutions updated to the cache too late. This would result in nodes not being pruned, but also redundant computation.

As for the considerably worse runtime performance, we consider two probable reasons. The first is the behavior just explained above. The second is that we require an exponential amount of tasks to be scheduled in this scenario (compared to only as many as CPU cores for the root-based one). While there are only as many tasks as threads at a time, there is still an overhead incurred from creating and managing them. We note that *std::async* is used for tasks. Its underlying implementation depends on the compiler used. For GCC, *std::async* creates a new *std::thread*, which is a native thread. LLVM has the same behavior. MSVC, on the other hand, uses Windows' concurrency API to create a thread pool which manages *std::async* calls. Our program was compiled using GCC. Switching to MSVC might reduce this overhead and yield competitive runtimes. Finally, we note that we previously used a naive thread pool implementation for task creation and management. This thread pool lacked any work-sharing or work-stealing features and did not result in significant improvements. Implementing a more complex thread pool tailored for our use case might result in noticeable speed-ups.

7.2 Ideal scenarios

As shown by the experiments, our method generally has a better performance than the state of the art. There are datasets for which its runtime is worse though, and these are datasets with small amounts of instances and features. These issues are also pronounced when running tests for $depth = 4$. This is because computing the solution for these kinds of input is very fast for sequential solutions. The overhead from creating threads and the

contention on shared resources cause this decrease in performance. The base time to compute the solution sequentially is so low that the time spent on ensuring synchronization exceeds it. Figure 2 shows this the best, as the more threads are used for some cases, the lower the speed-up is.

But, these are problems we can already solve very quickly with the already-existing methods. Therefore, we place much importance on results for "harder" problems. The same figure shows how the speed-up is increasingly larger for datasets with many instances and features. We consider *ionosphere* to be an outlier, and suspect that the reason why our method is slower for it (when searching for a complete tree) is that it has more features than instances, but this requires further investigation. Therefore, our method complements the state-of-the-art by allowing us to compute the ODT for larger, more complex datasets in feasible time.

7.3 Limitations

While our method generally outperforms the state of the art, it comes with a few limitations. Due to time and resource restrictions, we could not conduct the experiments on a different compiler or machine. As mentioned earlier, using MSVC instead of GCC might yield different results. Results for the leaf-based version might be considerably different, although we expect those for the root-based version to be similar. To ensure the optimality of our solutions, we compared the misclassification scores with those of STreeD. While this suggests that our solutions are correct, we do not have a formal proof to guarantee thread safety at this time. We also note that we cannot guarantee thread-safety specifically for partially ordered solutions (this means that the algorithm returns multiple solutions instead of just one). This is because we did not require synchronization on the upper bounds (UBs) in the case of totally-ordered solutions. That being said, fixing this is trivial, requiring locks on UB updates and retrieval. Due to time constraints, we could not fix this as it would have involved re-running all the experiments, although preliminary testing showed a minor difference in runtime performance.

7.4 Abstracting and Generalizing the Method

Our method can be easily applied to Murtree, considering that, for a large part, it shares the same algorithms and features as STreeD. That being said, we can extend it to other DP-based solutions. A base condition would be that there is a formula involving iteration over the features. Our shared memory model can be easily abstracted, any variables updated from multithreaded recursive calls would have to synchronized.

7.5 Applying P-STreeD to a Supercomputer

Machine Learning models nowadays (such as LLMs) may be trained on supercomputers. Using supercomputers for ODT computation would likely result in being able to use larger depths for even more complex problems or issues. Porting P-STreeD to a supercomputer requires adapting to MPI (Message Passing Interface) though. This involves transitioning to distributed memory parallelism. Each thread from our original approach would be instead represented by a MPI process (working on a subset of the features). To facilitate the adaptation of the shared memory model, a specific MPI process could be designated as a memory manager. It would handle requests from the other processes (through MPI routines) for operations involving the cache or solution. The cache would still require modifications

though, in order to address contention. An idea would be to keep two values for each entry, one of them would be synchronized and continuously updated. The other of them is given to the processes retrieving from it and is updated once an arbitrary time period with the value of the first. Alternatively, the cache could be made entirely local to each MPI process, with only the solution being shared.

8 Conclusions and Future Work

Dynamic programming solutions for Optimal Decision Tree building have seen notable advancements in recent years. As the demand for larger datasets and more complex problems continues to grow, improving the runtime for tree construction becomes increasingly important. One effective approach to enhance runtime is through multithreading, which leverages multiple CPU cores. However, applying multithreading is complex and carries risks, such as losing correctness and experiencing worse performance due to the need for synchronization.

To address these challenges, we proposed a multithreading approach for existing methods similar in concept to Murtree and STreeD. We divided features among the threads at the root level to compute their respective subproblems. We outlined a shared memory model for the solution and cache. We also investigated which commonly shared objects should be local to each thread. In our approach, each thread has its own two-terminal solvers and a similarity bound computer. We applied our method specifically to STreeD and compared two versions of when to start multithreading: close to the root level or the leaf nodes (the end of the recursion). We found that the root-based version tends to have fewer terminal calls, cache hits or misses, as well as better runtime performance than the leaf-based one. Comparing our method with DL8.5, Murtree, and STreeD, we observed generally faster runtimes, especially for larger datasets. We also investigated the speed-up for the number of threads employed and found that it is more pronounced for large datasets, with improvements exceeding 2.5 times in some cases when using 4 threads. However, for smaller datasets, the speed-up can be less significant, sometimes falling below 1, indicating worse performance than the baseline.

Despite these advancements, there is still future work to be done. We recommend investigating the use of a single cache for each thread instead of a shared cache. The implementation or integration of a threadpool into our current implementation should be explored. Further experiments should be conducted to compare the root and leaf versions using the MSVC compiler. We recommend examining the behavior of our method on specific datasets, such as *ionosphere*, where it was slower than its baseline. Finally, we recommend assessing the applicability of P-STreeD for supercomputers using MPI.

References

Aglin, G., Nijssen, S., & Schaus, P. (2020). Learning Optimal Decision Trees Using Caching Branch-and-Bound Search. In *Proceedings of AAAI-20* (pp. 3146–3153).

Agrawal, R., Mannila, H., Srikant, R., Toivonen, H. T., & Verkamo, A. I. (1996). Fast Discovery of Association Rules. In *Advances in Knowledge Discovery and Data Mining.*

Bertsimas, D., & Dunn, J. (2017, 07). Optimal classification trees. *Machine Learning*, *106*. doi: 10.1007/s10994-017-5633-9

Breiman, L., Friedman, J., Olshen, R., & Stone, C. (1984). Classification and Regression Trees. Taylor & Francis.

Costa, V., & Pedreira, C. (2023). Recent advances in decision trees: an updated survey. *Artificial Intelligence Review*, *56*, 4765–4800. doi: 10.1007/s10462-022-10275-5

Demirović, E., Lukina, A., Hebrard, E., Chan, J., Bailey, J., Leckie, C., ... Stuckey, P. (2022). MurTree: Optimal Decision Trees via Dynamic Programming and Search. *Journal of Machine Learning Research*, *23*(26).

Fromont, Ã., & Nijssen, S. (2007, 08). Mining Optimal Decision Trees from Itemset Lattices.. doi: 10.1145/1281192.1281250

Hu, X., Rudin, C., & Seltzer, M. (2019). Optimal Sparse Decision Trees. In *Advances in Neural Information Processing Systems* (Vol. 32). Curran Associates, Inc.

Hyafil, L., & Rivest, R. L. (1976). Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, *5*(1). doi: 10.1016/0020-0190(76)90044-9

Lin, J., Zhong, C., Hu, D., Rudin, C., & Seltzer, M. (2020). Generalized and Scalable Optimal Sparse Decision Trees. In *Proceedings of the 37th International Conference on Machine Learning (ICML-20)* (pp. 6150–6160).

Narlikar, G. J. (1998, December). *A Parallel, Multithreaded Decision Tree Builder* (Tech. Rep. No. CMU-CS-98-184). Pittsburgh, PA 15213: School of Computer Science, Carnegie Mellon University.

Narodytska, N., Ignatiev, A., Pereira, F., & Marques-Silva, J. (2018). Learning Optimal Decision Trees with SAT. In *International Joint Conference on Artificial Intelligence.*

Puthal, D., Wilson, S., Nanda, A., Liu, M., Swain, S., Sahoo, B. P., ... Prasad, M. (2022). Decision tree based user-centric security solution for critical IoT infrastructure. *Computers and Electrical Engineering*, *99*, 107754. doi: https://doi.org/10.1016/j.compeleceng.2022.107754

Quinlan, J. (1993). *C4.5: Programs for Machine Learning.* Elsevier Science.

Stivala, A., Stuckey, P. J., de la Banda, M. G., Hermenegildo, M., & Wirth, A. (2010). Lock-free parallel dynamic programming. *J. Parallel Distrib. Comput.*.

van der Linden, J., de Weerdt, M., & Demirović, E. (2023). Necessary and Sufficient Conditions for Optimal Decision Trees using Dynamic Programming. In *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)* (Vol. 36, pp. 9173–9212). (37th Annual Conference on Neural Information Processing Systems, NeurIPS 2023 ; Conference date: 10-12-2023 Through 16-12-2023)

Verwer, S., & Zhang, Y. (2019, Jul.). Learning Optimal Classification Trees Using a Binary Linear Program Formulation. *Proceedings of the AAAI Conference on Artificial Intelligence*, *33*(01), 1625-1632. doi: 10.1609/aaai.v33i01.33011624

A Experiments

A.1 Root versus leaf parallelization

Table 4: Comparison of statistics between multithreading versions for depth $d = 4$ and maximum number of feature nodes $n = 15$. $|D|$ and $|F|$ refer to the number of instances and features of the dataset, respectively. *Type* refers to where we apply the multithreading procedure; *none* is a version of our program with multithreading disabled. Time measured in seconds. *D2S Calls* denotes calls to the terminal solver. Best times marked with bold.

Dataset	$\|\mathcal{D}\|$	$\|\mathcal{F}\|$	Type	Time	D2S Calls	Cache Hits	Cache Misses
anneal	812	93	root	**<1**	869	88	2544
			leaf	**<1**	1127	115	3433
			none	**<1**	812	73	2360
diabetes	768	112	root	**<1**	4529	510	9527
			leaf	2	4432	613	11504
			none	2	4411	562	9138
heart-cleveland	296	95	root	**<1**	2497	201	5721
			leaf	**<1**	2370	282	6449
			none	**<1**	2247	231	5148
hepatitis	137	68	root	**<1**	795	26	1818
			leaf	**<1**	812	48	2106
			none	**<1**	697	33	1608
kr-vs-kp	3196	73	root	**<1**	993	40	2474
			leaf	**<1**	1035	61	2796
			none	**<1**	1000	36	2470
mushroom	8124	119	root	**<1**	238	1	616
			leaf	**<1**	545	6	1485
			none	**<1**	441	4	1206
pendigits	7494	216	root	**19**	6586	284	16585
			leaf	32	7971	278	21474
			none	51	7467	277	18147

Table 5: Comparison of statistics between multithreading versions for depth $d = 5$ and maximum number of feature nodes $n = 24$. $|D|$ and $|F|$ refer to the number of instances and features of the dataset, respectively. *Type* refers to where we apply the multithreading procedure; *none* is a version of our program with multithreading disabled. Time measured in seconds. *D2S Calls* denotes calls to the terminal solver. Timeouts of 600 seconds represented with $--$. Best times marked with bold.

Dataset	$\|\mathcal{D}\|$	$\|\mathcal{F}\|$	Type	Time	D2S Calls	Cache Hits	Cache Misses
anneal	812	93	root	**4**	20.588	20.990	87.808
			leaf	105	27.392	82.914	572.911
			none	6	18.287	19.776	75.086
diabetes	768	112	root	**35**	177.010	356.276	558.730
			leaf	238	179.604	624.795	1610.588
			none	71	176260	365.107	498.902
heart-cleveland	296	95	root	**9**	91.808	90.755	294.381
			leaf	111	90.974	125.710	853.270
			none	17	84879	83.333	256.523
hepatitis	137	68	root	$<$**1**	4.722	2.840	11.595
			leaf	2	3.702	3.437	21.106
			none	$<$**1**	3.542	2.416	9.141
kr-vs-kp	3196	73	root	**5**	15.834	6.693	59.548
			leaf	132	15.893	19.092	371.014
			none	10	14.334	6.080	53.680
mushroom	8124	119	root	4	5.771	621	18.761
			leaf	**2**	2.009	0.242	6.777
			none	3	2.002	0.236	6.303
pendigits	7494	216	root	**322**	137.341	38.893	377.413
			leaf	$--$	$--$	$--$	$--$
			none	$--$	$--$	$--$	$--$

A.2 State-of-the-art comparison

Table 6: Comparison of runtime performances between algorithms for depth $d = 4$ and maximum number of feature nodes $n = 15$. $|D|$ and $|F|$ refer to the number of instances and features of the dataset, respectively.

Dataset	$\|\mathcal{D}\|$	$\|\mathcal{F}\|$	DL8.5	STreeD	P-STreeD	Murtree
anneal	812	93	<1	<1	<1	<1
audiology	216	148	<1	<1	<1	<1
breast-wisconsin	683	120	<1	<1	<1	<1
diabetes	768	112	2	2	<1	2
fico-binary	10459	17	<1	<1	<1	<1
german-credit	1000	112	3	3	2	2
heart-cleveland	296	95	<1	<1	<1	<1
hepatitis	137	68	<1	<1	<1	<1
ionosphere	351	445	54	103	54	58
kr-vs-kp	3196	73	<1	<1	<1	<1
letter	20000	224	83	84	42	165
lymph	148	68	<1	<1	<1	<1
mushroom	8124	119	3	<1	<1	<1
pendigits	7494	216	24	39	20	47
splice-1	3190	287	61	211	106	134
tic-tac-toe	958	27	<1	<1	<1	<1
vehicle	846	252	6	9	5	7
yeast	1484	89	<1	<1	<1	<1

Table 7: Comparison of runtime performance between algorithms for depth $d = 5$ and maximum number of feature nodes $n = 24$. $|D|$ and $|F|$ refer to the number of instances and features of the dataset, respectively. Timeouts of 600 seconds represented with $--$.

Dataset	$\|\mathcal{D}\|$	$\|\mathcal{F}\|$	DL8.5	STreeD	P-STreeD	Murtree
anneal	812	93	5	6	3	4
audiology	216	148	9	<1	<1	<1
breast-wisconsin	683	120	12	9	5	5
diabetes	768	112	72	65	31	34
fico-binary	10459	17	<1	4	2	3
german-credit	1000	112	114	189	93	100
heart-cleveland	296	95	15	16	8	8
hepatitis	137	68	2	<1	<1	<1
ionosphere	351	445	--	--	--	272
kr-vs-kp	3196	73	12	11	5	12
lymph	148	68	2	<1	<1	<1
mushroom	8124	119	18	3	5	3
pendigits	7494	216	--	--	345	--
tic-tac-toe	958	27	<1	2	<1	2
vehicle	846	252	258	398	201	255
yeast	1484	89	43	30	13	19