Delft University of Technology

Research Project
CSE3000

Exploiting multi-core parallelism on optimal decision trees

Author
Ayush Patandin (4958195)

C

Responsible Professor & Supervisor
Emir Demirović

June 27, 2021

Abstract

This paper presents a study that discusses how multi-threading can be used to improve the runtime performance of constructing optimal classification trees. Decision trees are popular for solving classification or regression problems in machine learning. Heuristic methods are used to build decision tree algorithms that produce models of high accuracy within a short amount of time. An important limitation is that these heuristics locally optimize the decisions of the tree model. Consequently, in recent years, optimal classification tree algorithms have been introduced to strive for global optimality when learning decision trees. Unfortunately, the runtimes for constructing optimal decision trees are quite larger in comparison with the runtimes obtained from heuristic solutions. The study provides a mitigation for this by parallelizing the work of a recently invented optimal decision tree algorithm on multiple cores. There exist different parallel techniques to divide and schedule the work among processors. Our strategy follows the parallel approach that computes optimal decision trees using threads as processing elements in a shared memory space. In the end, we provide the experimental study to show that impressive runtime results of the optimal decision tree algorithm are successfully obtained with the help of the parallelization strategy.

1 Introduction

In modern machine learning, decision trees are widely used to break down a complex dataset into unique regions sequentially. Since decision trees have no such decision boundary that is a single or straight line, this type of data structure is considered as a non-linear classifier [1]. A decision tree is modeled by nodes that can be either leaves or branches. Every leaf node of the decision tree represents the class labels, while every branch node decides how the data should be split into smaller subsets.

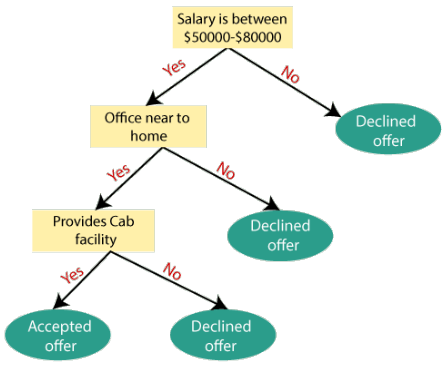


Figure 1: A decision tree for Job Offer Acceptance [2]

Going back to the 1970s, constructing optimal decision trees was proven to be a NP-hard problem [3]. Since the basic complexity of decision trees grow with their depth and their size, it is hard to compute decision trees of minimum size under the assumption of P $\neq$ NP. In practice, heuristic methods like CART [4] use an iterative approach to optimize decision trees based on a local objective function. Even though these heuristics can allow one to build decision trees of high accuracy within a small amount of time, there is no guarantee on whether these trees are of global optimality regarding factors such as their accuracy and their size. Due to this reason, optimal decision trees are more tempting to be used for producing models of high accuracy that generalise better on unseen data [5].

Some existing work already exists on producing accurate models using an optimal decision tree algorithm called MurTree [6]. This algorithm uses conventional algorithmic principles, such as dynamic programming and search, which is a memoization technique that allows reusing subproblems instead of recomputing them. This is useful for computing the optimal classification tree with a minimal misclassification score on the input dataset. By reusing intermediate computations of subtrees, the complexity of the algorithm is significantly better compared to naive exhaustive search.

Heuristic variants of parallel decision tree algorithms have already been exploited to a certain level in the past. One such example is the Communication-Efficient parallel algorithm [7] which makes use of the Parallel Voting Decision Tree (PV-Tree) data structure to tackle large communication costs. Despite the fact that MurTree [6] is very fast for an optimal decision tree algorithm, it still requires a lot more time compared to heuristic solutions. Even though the research into running parallel optimal decision trees is limited given the overall structure of the algorithm, it is reasonable to assume that parallelizing it on multiple cores is feasible. Exploring the topic is helpful, because it would incredibly speed up the generation of optimal decision trees.

Although optimal decision tree algorithms can be impressively optimized with the help of multi-core parallelism, it is not an easy task to simply divide and assign the work to available processors. Therefore, in order to optimize decision trees with the help of multi-core parallelism, one has to know some important things that build up the main problem of parallelizing the decision tree. At first, it is important to partition the incoming data from the root of the tree over the different child nodes. Since the child nodes each classify the data in a different way, this can be achieved by distributing the data in parallel over these nodes. Secondly, it is important to assign the tasks of the decision tree over different processes or threads, since the tasks done by the subtrees on the same layer of the decision tree are independent on each other. Thirdly, the processes or threads are mapped to logical processors or cores. So for these reasons, it is necessary to perform task parallelism and data distribution on the decision tree [8].

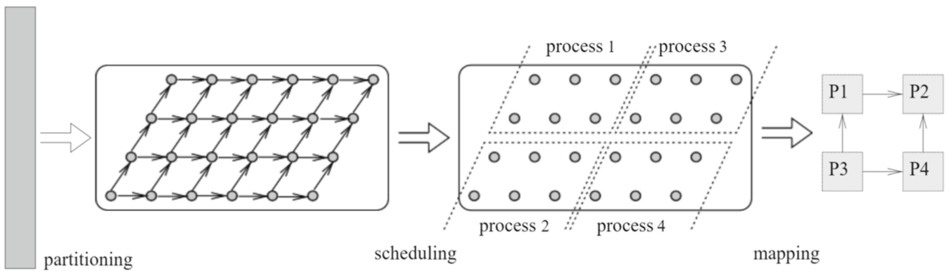


Figure 2: Parallelization of programs [9, p. 97]

Due to unequal work distributions, the possibility exists that there is less work done on one processor than on other processors. To tackle this, the workload needs to be scheduled more evenly to available cores such that the overall decision tree computation happens more efficiently. One scheduling technique is the concept of work sharing: a processor attempts to assign unfinished computations to other processors [10]. Another way of scheduling tasks is the concept of work stealing: some idle processor may decide to steal some of the tasks from other processors [10]. Both methods can reduce the runtime of the task computations. However, there is still a trade off in runtime, since communication overhead between the processors may cause some extra delay in the overall runtime of the program.

Contribution. The main problem this paper aims to address is stated as follows. *Optimal decision tree algorithms are usually sequential algorithms. Is it possible to exploit multi-core parallelism to produce optimal decision trees faster?*

The research question can be further divided in subquestions related to three different, yet equally important topics with regard to parallelizing decision trees.

1. What are the possible trade-offs that need to be investigated when parallelizing parts of an optimal decision tree algorithm?
   The use of more processors improves the runtime of constructing optimal decision trees but does not necessarily lead to an increase in program efficiency, since there will be less computations per available worker. Therefore, a study is important to show the growths of the speedup and the efficiency for different numbers of CPUs.

2. Which approach can be used to split the data and the tasks in parallel among the different workers in the decision tree?
   The problem size of the program is identified by the number of features in the input dataset, therefore the computations for these attributes should be partitioned and scheduled among the available workers.

3. How can correctness and performance issues be avoided when integrating parallelism into a decision tree algorithm where most of the tree computation and data distribution happen independently on the different decision tree nodes?

This subtask requires to identify the possible setbacks that may add extra delay to the program or that may cause the program to produce unexpected results on information of the decision tree. For example, it is needed to remove unnecessary data dependencies between variables that depend on each other in order for parallelization to take part in the code block where those variables are used.

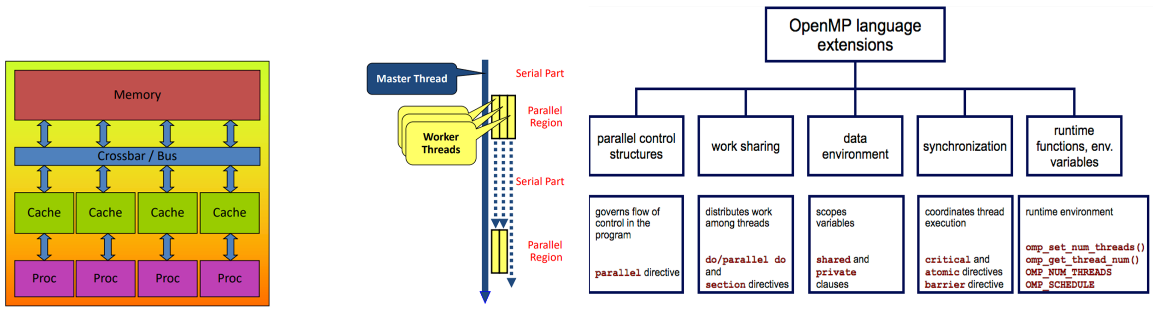
First, this paper serves which method will be used for parallelizing explicit parts of the MurTree algorithm. Second, the preliminary work of implementing the sequential program will be discussed. The next section describes the parallelization strategy which is used for this research. Then, the results and argumentations will be given of how the parallel program performs compared to the sequential program. Thereafter, this paper continues with a section on responsible research that reflects on the ethical aspects of this research and how its methods can be reproduced. This paper ends with the conclusions and recommendations of this research.

2 Methodology

In order to accomplish the aims set in the previous section, this section provides some remarkable approaches that can help to parallelize the work of optimal decision trees. Furthermore, this section argues which method is appropriate to use for the MurTree algorithm and what can be expected from such a method.

By applying a parallel approach on the MurTree algorithm, multiple processors will then be able to execute specific parts of the program that use a magnitude of computations. The complexity of the MurTree algorithm mainly grows with a general depth while decision trees of a maximum depth of two are computed fast even for large datasets. Consequently, it would be more beneficial to parallelize the parts of the MurTree algorithm that construct optimal trees with an arbitrary depth. Hence, trees that contain many feature nodes would be built in parallel among multiple processing units and scalable optimizations would then be guaranteed in the bottleneck of the algorithm.

Decision trees can be optimized with the help of the multithreaded API, OpenMP: each thread has access to its own chunk of data through a shared memory system [11]. Programming is done by adding directives and the parallelization is done by the compiler. The sequential program starts off with a single thread (master thread), followed by different parallel regions which each make use of a team of threads that execute instructions concurrently.



(a) OpenMP Shared-Memory Fork-Join Model [12]

(b) OpenMP core elements [13]

Figure 3: OpenMP architecture

There are different ways to schedule threads. For each schedule, it is possible to specify the chunk size, that is the maximal amount of data assigned to each thread. Threads can be scheduled statically such that blocks of fixed size are assigned to the available workers in round-robin fashion. Threads can also be scheduled dynamically, meaning that blocks are only assigned to threads that have finished processing their previous blocks.

The use of multi-threading can also bring some pitfalls as is not a trivial solution for optimizing decision trees. One such pitfall is a data race: if multiple threads modify some element at the same time, then this may result in unpredictable solutions (race condition) [14]. The MurTree algorithm updates the shared variable T_{best} to find the best optimal tree, and therefore, synchronization can be used in the parallel region of the algorithm to mutually exclude threads from accessing such a global element simultaneously. Another issue is when a cache line is being accessed by multiple threads or when a cache line is being altered by some thread. It takes some delay to transfer the same line between threads due to false sharing [15]. This performance issue can be avoided by padding the cache lines such that each line only stores one variable. It would be possible to pad the entries on different cache lines to slightly increase the parallel performance of the program, since the MurTree algorithm makes use of a shared cache and updates its cache entries using the similarity-based lowerbound [6, sec. 4.5].

Tasks can be assigned to different processes to divide the work of the MurTree algorithm with the help of Open MPI [16]: each process performs the computations for its assigned block of features by executing its own part of the program and exchanges the information of the local best decision trees with other processes by sending and receiving messages. The program recognizes the processes by their process rank during execution. In contrast to threads, processes do not use a shared memory space. The local variables, i.e. the feature and the possible subtree size, used for solving optimal decision trees are only accessible within the scope of the current worker. Since each process has its separate main memory block, point-to-point communication can take place through message passing over the network.

For such a messaging protocol, it is important to prevent possible setbacks that can occur in the program. For instance, when two or more processes simultaneously send the misclassification score of their computed subtree over the network, each one of them will wait for a response before constinuing with communication. This may lead to a deadlock [17]. So the order of blocking send and receive operations matter: the workers should first retrieve information on their block of assigned features before they start to perform work on their part of the algorithm and send their computed decision tree results to the main process.

Both OpenMP [11] and OpenMPI [16] can be used to parallelize an application, however, one should be able to recognize which parallelization strategy would be the most suitable approach to use for the MurTree algorithm [6]. By parallelizing the decision tree with OpenMPI [16], the best splitting attribute can be found quite fast, since the processes can communicate with each other by sending the information of their local computation to one another. For example, the optimal decision tree with its minimal misclassification score can be determined by assigning different splitting attributes to each process. Each process will then compute the local best decision tree using its assigned chunk of splitting attributes. By sending over the information of local best decision trees through message passing, the best decision tree can be solved more efficiently. On the other hand, by making use of the parallel directives of OpenMP [11], there is no explicit communication required between the threads, since they share the same memory space. So each thread would find its own local best decision tree in the parallel region. Among the worker threads, the final best decision tree which gets assigned to the master thread is the local best decision tree with the smallest misclassification score. Since there is limited information in the MurTree algorithm [6] that needs to be passed over from one processing element to another, OpenMP [11] seems to be the better parallel approach for this study.

The main expectation of using the OpenMP implementation tool [11] is that the program should become more efficient in terms of performance. This means that, when adding the parallel directives, one should prevent the possibility that the program becomes slow due to an imperfect load balance of the work among the threads. The goal is to use a suitable amount of threads such that there exists an equal distribution over the entire workspace while, at the same time, there are no threads that remain idle. Therefore, the entire chunk of tasks of the decision tree should be evenly split over the worker threads such that the efficiency of the program is kept high as each processing element is occupied with its own assigned working block.

3 Preliminary work

This section describes the implemented work of the MurTree algorithm [6] which has been used to construct optimal decision trees. First, a brief idea is given on how the main program works. Next, the algorithms which have been implemented from the MurTree paper [6] are discussed. These algorithms include the specialized depth-two algorithm and the general depth algorithm for constructing optimal decision trees. Moreover, this section provides metrics for computing the error of the decision tree model and for approximating the performance when running the program in parallel.

3.1 Main Program Description

The main program outputs a decision tree representation with the minimum number of misclassifications possible with as input a dataset file together with the user-specified decision tree characteristics. One provided user input is the maximum depth which tells the program what should be the maximum level of feature nodes in the decision tree. Another input which the user can give to the program is the maximum number of feature nodes that the decision tree is allowed to use to classify the data going through those nodes.

The user should only input dataset files that are fully binarised, which means that each instance is only filled with zeros and ones. The program considers the first column of the dataset to be the target variable. The program also allows the user to load dataset files that include attribute names in their first row, however, default labels are assigned to the columns if no attribute names are specified by the user.

3.2 Algorithms used

The MurTree algorithm [6] already has existing work on the specialized depth-two algorithm and the general depth algorithm, which have been implemented for this research paper. In the sections below, a detailed description will be given on how the depth-two algorithm and the general depth algorithm have been approached for this research.

3.2.1 Specialized depth-two algorithm

The first phase of the specialized depth-two algorithm computes all the individual frequency counts of single features and the pairwise frequency counts of combined features in the input dataset. Since the dataset is fully binarised, $|D^+|$ represents the amount of positive instances of the dataset, while $|D^-|$ represents the amount of negative instances of the dataset. For given features f_i and f_j, $FQ^+(f_i)$ gives the individual counts of the positive instances and $FQ^+(f_i, f_j)$ give the pairwise counts of the positive instances. Individual and pairwise frequency counts for negative instances are denoted similarly with a minus sign: $FQ^-(f_i)$ and $FQ^-(f_i, f_j)$. Besides instances, features are also represented as positive (f_i) and negative $(\overline{f_i})$ features. Positive features take zero as their value and negative features take one as their value.

In algorithm 1, the individual and pairwise frequency counts are used to compute the frequency counts $FQ^+(\overline{f_i}, f_j), FQ^-(\overline{f_i}, f_j), FQ^+(\overline{f_i}, \overline{f_j})$ and $FQ^-(\overline{f_i}, \overline{f_j})$. The frequency counting rules in section 4.4.1 of the MurTree paper [6] have been applied for the first phase, while the rules for computing the misclassification scores of the left and right subtree are part of the second phase and can be found in section 4.4.2 of the MurTree paper [6].

In the last phase of algorithm 1, the optimal trees with a maximum depth of two and with a maximum size of three are constructed using attributes from the best left and right subtrees together with the computed frequencies of the dataset D. The possible parameters of leaf nodes are 1 to represent classifications of positive instances and 0 to represent classifications of negative instances. Feature nodes, on the other hand, take a feature, a left child node and a right child node as their parameters.

3.2.2 General depth algorithm

The algorithm for constructing optimal decision trees with a general depth can be found in al. 2. First, the base cases are given, followed by the general case of the decision tree algorithm. The tree depth and the tree size should be given as positive input numbers to the algorithm in order to result in a valid output.

If the number of nodes is greater than $2^d - 1$, then the parameter n can be reduced to this size, because a perfect binary tree [18] can only have this amount as the maximum number of nodes, that is, a decision tree where each feature node has exactly two children and where every leaf node is present at the last layer of the data structure. As it is not possible to construct a tree where its depth is greater than its size, the tree size can be reduced to the tree depth whenever this is the case.

Besides that, a search through the cache takes place to find if there already exists some optimal subtree for the given branch. The algorithm stores each computed subtree with its misclassification score in some hash table. In order to represent a branch as an integer, the hash table makes use of the hashfunction given in algorithm 5 from section 4.6.1 of the MurTree paper [6]. Each entry

Algorithm 1 Algorithm for computing optimal decision trees with a maximum depth of 2

```
1: Input: Binary dataset $\mathbf{D}$
2: Output: The optimal decision trees with a max depth of 2 and with max 3 feature nodes
   together with their corresponding misclassification score.
3: begin
4: $\forall f_i : FQ^+(f_i) \leftarrow 0, \quad \forall f_i : FQ^-(f_i) \leftarrow 0$
5: $\forall f_i, f_j : FQ^+(f_i, f_j) \leftarrow 0, \quad \forall f_i, f_j : FQ^-(f_i, f_j) \leftarrow 0$
6: for Record $R \in \mathbf{D}$ do
7:    if $targetClass(R) = 1$ then
8:       increment $FQ^+$
9:    else
10:      increment $FQ^-$
11:   end if
12:   for $f_i \in R$, predicate$(f_i)$ = true do
13:      if $targetClass(R) = 1$ then
14:         increment $FQ^+(f_i)$
15:      else
16:         increment $FQ^-(f_i)$
17:      end if
18:   end for
19:   for $f_i \in R$ , $f_j \in R, i < j$, predicate$(f_i)$ = true && predicate$(f_j)$ = true do
20:      if $targetClass(R) = 1$ then
21:         increment $FQ^+(f_i, f_j)$
22:      else
23:         increment $FQ^-(f_i, f_j)$
24:      end if
25:   end for
26: end for
27: for $f_i \in featuresOf(\mathbf{D})$ do
28:    for $f_j \in featuresOf(\mathbf{D})$, $i \neq j$ do
29:       $FQ^+(\overline{f_i}, f_j) \leftarrow FQ^+(f_j) - FQ^+(f_i, f_j)$
30:       $FQ^-(\overline{f_i}, f_j) \leftarrow FQ^-(f_j) - FQ^-(f_i, f_j)$
31:       $FQ^+(\overline{f_i}, \overline{f_j}) \leftarrow |D^+| - FQ^+(f_i) - FQ^+(f_j) + FQ^+(f_i, f_j)$
32:       $FQ^-(\overline{f_i}, \overline{f_j}) \leftarrow |D^-| - FQ^-(f_i) - FQ^-(f_j) + FQ^-(f_i, f_j)$
33:       $CS(\overline{f_i}, f_j) \leftarrow \mathbf{min}(FQ^+(\overline{f_i}, f_j), FQ^-(\overline{f_i}, f_j))$
34:       $CS(\overline{f_i}, \overline{f_j}) \leftarrow \mathbf{min}(FQ^+(\overline{f_i}, \overline{f_j}), FQ^-(\overline{f_i}, \overline{f_j}))$
35:       $MS_{left}(f_i, f_j) \leftarrow CS(\overline{f_i}, f_j) + CS(\overline{f_i}, \overline{f_j})$
36:       if bestLeftSubTree[$f_i$].MS > $MS_{left}(f_i, f_j)$ then
37:          bestLeftSubTree[$f_i$].MS $\leftarrow MS_{left}(f_i, f_j)$
38:          bestLeftSubTree[$f_i$].feature $\leftarrow f_j$
39:          bestLeftSubTree[$f_i$].classifyRight $\leftarrow$ if$(FQ^+(\overline{f_i}, f_j) > FQ^-(\overline{f_i}, f_j))$ then 1 else 0
40:          bestLeftSubTree[$f_i$].classifyLeft $\leftarrow$ if$(FQ^+(\overline{f_i}, \overline{f_j}) > FQ^-(\overline{f_i}, \overline{f_j}))$ then 1 else 0
41:       end if
42:       $FQ^+(f_i, \overline{f_j}) \leftarrow FQ^+(f_i) - FQ^+(f_i, f_j)$
43:       $FQ^-(f_i, \overline{f_j}) \leftarrow FQ^-(f_i) - FQ^+(f_i, f_j)$
44:       $CS(f_i, \overline{f_j}) \leftarrow \mathbf{min}(FQ^+(f_i, \overline{f_j}), FQ^-(f_i, \overline{f_j}))$
45:       $CS(f_i, f_j) \leftarrow \mathbf{min}(FQ^+(f_i, f_j), FQ^-(f_i, f_j))$
46:       $MS_{right}(f_i, f_j) \leftarrow CS(f_i, \overline{f_j}) + CS(f_i, f_j)$
47:       if bestRightSubTree[$f_i$].MS > $MS_{right}(f_i, f_j)$ then
48:          bestRightSubTree[$f_i$].MS $\leftarrow MS_{right}(f_i, f_j)$
49:          bestRightSubTree[$f_i$].feature $\leftarrow f_j$
50:          bestRightSubTree[$f_i$].classifyRight $\leftarrow$ if $(FQ^+(f_i, f_j) > FQ^-(f_i, f_j))$ then 1 else 0
51:          bestRightSubTree[$f_i$].classifyLeft $\leftarrow$ if $(FQ^+(f_i, \overline{f_j}) > FQ^-(f_i, \overline{f_j}))$ then 1 else 0
52:       end if
53:    end for
54: end for
55: T(D, n=0), T(D, n=1), T(D, d=2, n=2), T(D, d=2, n=3) $\leftarrow constructDepth2Trees()$
    using
56:       $argmin_{f_i \in \mathbf{D}}$ (bestLeftSubTree[$f_i$].MS + bestRightSubTree[$f_i$].MS)
57: return T(D, n=0), T(D, n=1), T(D, d=2, n=2), T(D, d=2, n=3)
```

Algorithm 2 Algorithm for computing optimal decision trees with a general depth

```
1: Input: Binary dataset $\mathbf{D}$, max. depth $\mathbf{d}$, max. number of nodes $\mathbf{n}$, memoisation cache mem, path from the root to the subtree branch
2: Output: The best decision tree with general depth and number of nodes together with its corresponding misclassification score: T(D, d, n, mem, branch)
3: begin
4: if $\mathbf{d} < 0$ or $\mathbf{n} < 0$ then
5:     return $\emptyset$
6: end if
7: if $\mathbf{n} > 2^{\mathbf{d}} - 1$ then
8:     return T(D, d, $2^{\mathbf{d}} - 1$, mem, branch)
9: end if
10: if $\mathbf{d} > \mathbf{n}$ then
11:     return T(D, n, n, mem, branch)
12: end if
13: T $\leftarrow$ FindOptimalSubTreeInCache(cache, mem, n)
14: if T $\neq \emptyset$ then
15:     return T
16: end if
17: if $\mathbf{d} \leq 2$ then
18:     T $\leftarrow$ ComputeOptimalTreeWithSpecializedDepth2(D)
19:     PruneOptimalTree(T)
20:     AddSolutionToCache(mem, T, branch, n)
21:     return T
22: else
23:     $n_{max} \leftarrow \mathbf{min}(2^{\mathbf{d}-1} - 1, \mathbf{n} - 1)$,    $n_{min} \leftarrow (\mathbf{n} - 1 - n_{max})$
24: end if
25: $T_{best} \leftarrow \emptyset$
26: $MisclassificationScore(T_{best}) \leftarrow \infty$
27: $NumberOfFeatureNodes(T_{best}) \leftarrow \mathbf{n}$
28: #pragma omp parallel for private($f_i, n_R, n_L$) collapse(2) reduction(MS_min : $T_{best}$)
29: for $f_i \in featuresOf(\mathbf{D})$ do
30:     for $n_R \in$ range$(n_{min}, n_{max})$ do
31:         $n_L \leftarrow \mathbf{n} - 1 - n_R$
32:         $\mathbf{D}_L \cup \mathbf{D}_R \leftarrow SplitDataSetOnFeature(\mathbf{D}, f_i)$
33:         $B_L \leftarrow$ branch,    $B_R \leftarrow$ branch
34:         $B_L.addBranchIndex(2i)$,    $B_R.addBranchIndex(2i + 1)$
35:         $T_L \leftarrow \mathbf{T}(D_L, \mathbf{d - 1}, n_L, \mathbf{mem}, B_L)$
36:         $T_R \leftarrow \mathbf{T}(D_R, \mathbf{d - 1}, n_R, \mathbf{mem}, B_R)$
37:         $T_{localbest} \leftarrow FeatureNode(f_i)$
38:         $LeftSubTree(T_{localbest}) \leftarrow T_L$
39:         $RightSubTree(T_{localbest}) \leftarrow T_R$
40:         $MisclassificationScore(T_{localbest}) \leftarrow MisclassificationScore(T_L) + MisclassificationScore(T_R)$
41:         $NumberOfFeatureNodes(T_{localbest}) \leftarrow 1 + NumberOfFeatureNodes(T_L) + NumberOfFeatureNodes(T_R)$
42:         $T_{best} \leftarrow TreeWithSmallestMisclassificationScore(T_{best}, T_{localbest})$
43:     end for
44: end for
45: AddSolutionToCache(mem, $T_{best}$, branch, n)
46: return $T_{best}$
```

can be retrieved by a path from the root to the branch node of the optimal subtree. If such a path has been identified in the hash table, then the algorithm does not need to perform any other computations and returns this subtree as its found solution.

For a depth of at most two, the specialized depth-two algorithm in al. 1 is used to build the optimal tree. In addition, the retrieved tree gets pruned to a smaller tree if there are redundant tree components that can be removed to improve the accuracy by the reduction of overfitting. If a smaller tree has a smaller misclassification score, then the current tree gets pruned to that smaller tree. Thereafter, the algorithm stores the depth-two tree in cache and returns the result.

For the general case, the algorithm starts off with an uninitialized tree with an infinite amount of misclassifications. For each feature f_i with index i, the algorithm goes through the possible tree sizes $[n_{min}, n_{max}]$ and constructs the local best decision tree with its left and right subtrees recursively. The values $2i$ and $2i+1$ are added to the branch of the left and right subtrees respectively, as discussed in section 4.6.1 of the MurTree paper [6]. Each local best tree takes a misclassification score which is the sum of the misclassification scores of its children and the best tree is the tree which has the minimum misclassification score. Lastly, the final solution is added to the cache and is returned by the algorithm.

3.3 Accuracy and Performance Metrics

To validate how well decision trees work on different binarised datasets, there is a significant amount of research done on parts that can help the program to perform experiments on the MurTree algorithm. Firstly, the input dataset is partitioned in two different sets given below.

- The train set is used for learning the decision tree model. The aim is to generate a good predictive model that fits as many records as possible when building the decision tree. The train set should have a sufficient large size such that there are enough known records for which a trained decision tree model can be produced.
- The test set is used for evaluating the decision tree constructed by the train set. The decision tree model which fits the train set should also be able to perform well on the data of the test set. Otherwise, overfitting occurs when the model does not generalize well on any unseen data other than the data from the train set.

After building the decision tree on the data from the train set, it is evaluated how many records from the test set are misclassified on the constructed decision tree model.

$$\mathrm{MS}_{test} = |\mathrm{S}| \tag{1}$$

with $\mathrm{S} = \{R \in \mathrm{D}_{test} \quad | \quad T.classify(R) \neq targetClass(R)\}$.
Here, S represents the set of records in the dataset that are not classified to their target class. The errors measured during the experiments are divided by the number of records in the test set to normalize the misclassification score to a value between 0 and 1.

$$\varepsilon_{test} = \frac{\mathrm{MS}_{test}}{|\mathrm{D}_{test}|} \tag{2}$$

On the contrary, the test accuracy equals $1 - \varepsilon_{test}$ and measures the correctness in terms of generalisation of the decision tree model, that is, how well the decision tree performs on the test set.

There are several performance metrics used for parallel programs [9, ch. 4.2]. Firstly, the serial execution time T_s measures the total duration for which the entire program is executed sequentially. For this metric, the computations are processed one at a time. Secondly, the parallel execution time T_p measures the total duration in which p processors are used to execute the entire program. This duration goes on until every processor has finished executing their part of the program.

The user can set the number of threads by changing the OpenMP environment variable `$OMP_NUM_THREADS`. The performance of the program depends on the available amount of cores and logical processors on the user's local machine. It is possible to queue up more threads than logical processors p, however, the operating system would then only be able to schedule p threads at the same time, leaving the remaining threads unoccupied. Therefore, setting the number of omp threads to the number of logical processors would give the best obtainable runtime for executing the parallel program.

The aim is to distinguish differences in runtime between T_s and T_p. These differences are usually expressed as speedups which give an indication on how much faster the sequential program works with the use of parallelism. There are two different approaches on how to measure the speedup for some parallel program.

1. True speedup: the fraction that divides the best serial execution time by the parallel execution time [19, p. 5].
$$S_p = \frac{T_s}{T_p} \tag{3}$$

2. Relative speedup: the fraction that divides the execution time of a single processor by the parallel execution time [20].
$$S_p = \frac{T_{p=1}}{T_p} \tag{4}$$

In practice, the parallel runtimes are often compared to the runtime obtained by a single master thread. Therefore, from this point onward, the relative speedup is used as the speedup metric for this research.

Another way to assess the performance of parallel programs is by measuring their efficiency. For this metric, it is determined how well a processor performs on its assigned block of tasks. The formula for computing the efficiency is defined in the equation below.

$$E_p = \frac{S_p}{p} = \frac{T_{p=1}}{p \cdot T_p} \tag{5}$$

This formula shows how much time a processor uses to finish the same tasks that are also performed by the sequential program.

4 Parallelization Strategy

For the main program discussed in the previous section to be able to run with multi-threading, the OpenMP directives [11] from section 2 have been used on the most computationally expensive part of the MurTree algorithm [6]. The largest bottleneck of this algorithm lies in the nested for loops of the general depth algorithm (lines 29-30 of al. 2). The main reason that a higher runtime occurs in this part of the algorithm is that, for each predicate, different tree sizes are used for the left and the right optimal subtrees to construct the root of the decision tree.

The performance of the program has been improved by partitioning the tasks of the nested for loops in chunks where each thread is assigned to its own chunk of tasks. Dynamic scheduling with varying small chunk sizes have been considered, however, it entailed much overhead as there would be a greater amount of schedules from blocks to threads. There are more schedules needed if a few tasks is assigned to each thread. On the other hand, the efficiency of the program decreases if there is a great number of threads that remain idle over time. Therefore, the default way of scheduling the threads is `schedule(static)` with a fixed chunk size and seemed to be the most effective optimization to assign chunks of tasks to threads.

$$\texttt{Chunk size} = \frac{\texttt{L}}{\texttt{T}} \tag{6}$$

with $\texttt{L}$ = number of loop iterations and $\texttt{T}$ = number of threads. If there were to be a remainder of loop iterations, then these tasks would be assigned to the first $\texttt{L} \pmod{\texttt{T}}$ threads.

The OpenMP core elements in line 28 of al. 2 are used for building the parallel region. Each feature f_i is not shared with more than one thread. This variable is only used for the local loop iteration, therefore it is made private. The same reasoning also holds for other private fields such as the left tree size n_L and the right tree size n_R.

The problem of having idle threads occurs when there is no equal load balance of the work. For instance, benchmarks with a small amount of features may not perform well for multiple threads, because some threads will be wasted as there are no tasks left to fulfill. As a solution, the OpenMP collapse clause [21] is used to increase the total number of iterations for the number of available threads. In other words, the tasks of the outer and inner for loops (lines 29-30 of al. 2) have been collapsed in one large chunk before the partitioning happens. This reduces the granularity of the work each thread needs to perform. Therefore, the parallel region of the algorithm is more scalable than it was before.

The idea of avoiding a data race between threads is that threads should not update a shared variable, i.e. T_{best}, at the same time in the parallel region of the MurTree algorithm. To avoid this from happening, such an attribute can be reduced by applying the OpenMP reduction clause [22]. For this clause, the reduction operator is applied on the best optimal decision trees, where each thread computes the local T_{best} in its own workspace. For each thread, T_{best} is reduced to the best decision tree with the smallest misclassification score. Since the reduction is user-defined, the `omp declare reduction` directive [23] is used to declare the reduction identifier `MS_min` in the reduction clause. The used declaration is expressed in Equation 7. The first parameter represents the identifier of the reduction operator. The second parameter specifies the type of the variable that should be reduced. In this case, the type of T_{best} is provided to omp. At last, the behaviour of the reduction operator is specified. The operator behaves according to the function `TreeWithSmallestMisclassificationScore` (line 42 of al. 2), which takes two decision trees and returns the decision tree with the smallest number of misclassifications. The initializer clause specifies that the program should start its reduction from the original variable which is provided to omp. The decision trees, computed by each worker thread, are passed through the reduction operator and the final result is assigned to the master thread.

$$
\begin{aligned}
&\#\text{pragma omp declare reduction(}\\
&\qquad \texttt{MS\_min :}\\
&\qquad \mathit{typeOf}(T_{best}) :\\
&\qquad \text{omp\_out}{=}\mathit{TreeWithSmallestMisclassificationScore}(\text{omp\_out, omp\_in}))\\
&\quad \text{initializer(omp\_priv}{=}\text{omp\_orig)}
\end{aligned}
\tag{7}
$$

All in all, the MurTree algorithm has been successfully optimized with the use of omp directives and the high accuracy of the program has been maintained without any side effects after integrating the parallel OpenMP directives into the MurTree algorithm. In the next section, the runtime improvements will be shown for different numbers of threads and for binarised datasets with different numbers of features.

5 Experimental Study

This section has the purpose to show a decrease in misclassification score for decision trees with a growing depth, to show that the parallel program ensures better runtime results compared to the serial program as well as to argue how well decision trees generalise for different benchmarks. On another note, this section investigates the trade-off between the speedup and the efficiency of the program for multiple processors.

First, this section gives some insight on the experimental setup of the parallel MurTree algorithm. Then, this section presents the gathered results for the performed experiments together with a discussion on the accuracies and the differences in runtime between the sequential and the parallelized program.

5.1 Experimental setup

For testing the accuracy and the performance of the decision tree algorithm, the two main experiments that have been done on a great number of binarised dataset files are the average optimal tree experiment and k-fold cross validation [24]. The subsections below describe how each experiment works and what each experiment measures.

5.1.1 K-fold cross validation

This experiment requires the records of the dataset to be shuffled and partitioned in k folds. For each fold, a single experiment takes place. The current fold is then used as test set, while the other k − 1 folds are merged into one train set. Next, the train set is used to model the optimal decision tree and the error is measured on the test set analogously as discussed in Equation 2.

Once the single experiments are completed, the mean is computed on the accuracies of the k folds. The standard deviation is also computed to give an indication whether the decision tree model has been generalized well enough, since for each fold, a different part of the dataset is taken as unseen data. So the accuracies should be close to each other to prove that the model fits well on different folds which are used as test sets.

5.1.2 Average optimal tree experiment

For this experiment, the optimal decision tree algorithm is run for a fixed number of times. For each run, a fixed splitting fraction is used to divide the input dataset in a train and a test set. The records that are placed in the test set are taken randomly from the main dataset, and thereafter, the remaining records from the main dataset are added to the train set.

Once the partitioning is done, the program measures the runtime of the algorithm to construct the optimal decision tree on the train set and computes the accuracy of the decision tree model on the test set using ε_{test} from Equation 2. For each run, the runtime and the accuracy for constructing the current optimal decision tree are separately aggregated to their sum. At the end of the experiment, the average performance and the average test accuracy can be measured by dividing the sums of the runtime and the accuracy by the total number of runs.

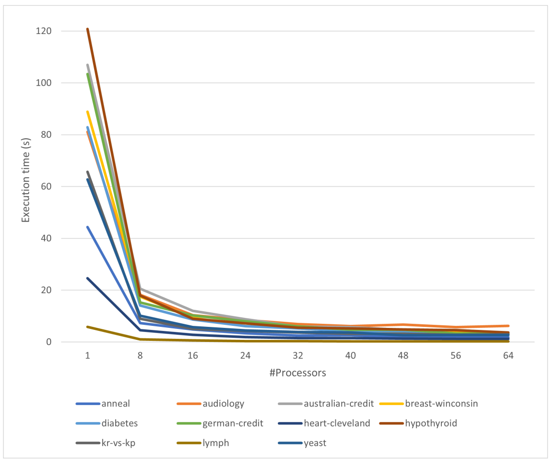
5.2 Results and discussions

In Table 1, single tree experiments are performed on 25 binarised datasets from which the number of instances and the number of features are given. For this table, the datasets are entirely used to train the decision tree model. The table gives an overview of the results for decision trees of $depth \in [1, 2, 3, 4]$. These results show the number of instances that are misclassified by the decision tree. The expected observation is that decision trees of larger depth have a smaller misclassification score compared to decision trees of smaller depth. This proves that the algorithm constructs more nodes on a larger decision tree such that more instances are correctly classified to their target class.

Name	$\|D\|$	$\|F\|$	MS$_{depth=1}$	MS$_{depth=2}$	MS$_{depth=3}$	MS$_{depth=4}$
anneal	812	93	151	137	112	91
audiology	216	148	29	10	5	1
australian-credit	653	125	89	87	73	56
breast-wisconsin	683	120	48	22	15	7
compas-binary	6907	12	2494	2333	2272	2250
diabetes	768	112	196	177	162	137
fico-binary	10459	17	3180	3019	2959	2894
german-credit	1000	112	290	267	236	204
heart-cleveland	296	95	69	60	41	25
hepatitis	137	68	19	16	10	3
hypothyroid	3247	88	118	70	61	53
ionosphere	351	445	59	32	22	7
kr-vs-kp	3196	73	1012	418	198	144
letter	20000	224	813	599	369	261
lymph	148	68	30	22	12	3
mushroom	8124	119	920	252	8	0
pendigits	7494	216	505	153	47	13
primary-tumor	336	31	70	58	46	34
segment	2310	235	41	9	0	0
soybean	630	50	92	55	29	14
splice-1	3190	287	575	508	224	141
tic-tac-toe	958	27	288	282	216	137
vehicle	846	252	189	75	26	12
vote	435	48	19	17	12	5
yeast	1484	89	442	437	403	366

Table 1: Single tree experiments with dataset properties and misclassification score for different tree depths.

In Figure 4a, the execution times T_p are measured in seconds. As expected, it can be observed that for each dataset, an increase in processors shows a better performance in runtime. For a large amount of processors, there are less computations that need to be performed by each individual thread, hence the execution time becomes smaller for training the decision tree model on different benchmarks.

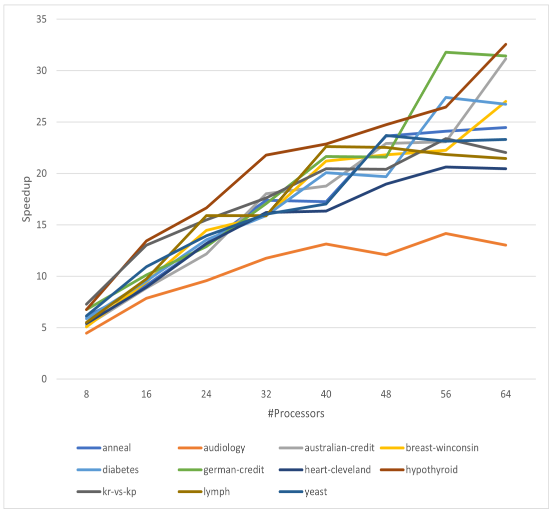


(a) Parallel execution time T_p of single tree experiments for different number of processors

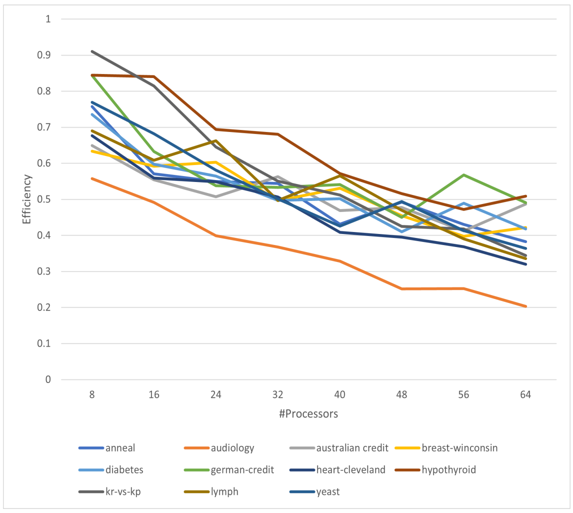
Figure 4: Performance metrics for constructings decision trees of $depth = 4$

Figure 4b and Figure 4c show the measured speedup and efficiency that are computed with the formulas in Equation 4 and in Equation 5 respectively. In all of the cases, the speedups are sublinear [25], which means that $S_p < p$. If the amount of processors increases, then the speedup also increases, but at a smaller rate until it converges to the maximal speedup, which is obtained for the maximum number of available CPUs on the running machine.

The program becomes less efficient for more processors, since there are more workers required to perform the same amount of computations that can be processed by the sequential program. The fraction between the speedup S_p and the number of processors p becomes smaller, because S_p does not grow proportionally with p, but instead grows a bit slower. The use of more processors to perform the same amount of work does lead to a better runtime, but requires more workers, and therefore leads to a less efficient program.



(b) Speedup S_p of single tree experiments for different number of processors



(c) Efficiency E_p of single tree experiments for different number of processors

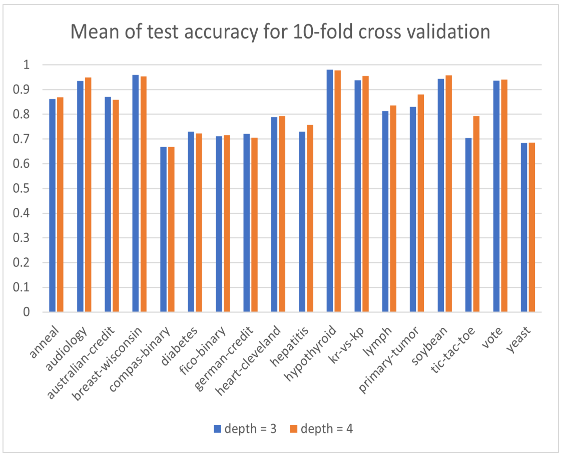
Figure 4: Performance metrics for constructings decision trees of $depth = 4$ (cont.)

The benchmark results for measuring the mean and the standard deviation of the test accuracy are displayed together with their parallel execution times T_p for $depth \in [3, 4]$ in Figure 5 using 10-fold validation, which follows the procedure from section 5.1.1.

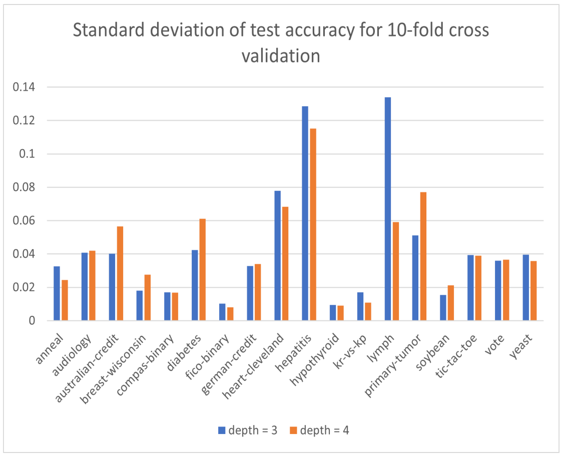
The majority of the benchmarks obtain test accuracies that lie above 0.7, which means that the decision tree model generalizes well on the unseen data from these files. The standard deviation seems to be high for benchmarks that have a relatively small number of instances. For instance, hepatitis and lymph both use less than 200 records, thus each fold uses a limited size of unseen data. As it stands, there are more datasets for which their decision tree model of depth 4 generalizes strictly better compared to their decision tree model of depth 3. With regards to the aggregated runtime of the experiment, each fold takes more time to construct trees of depth 4 compared to trees of depth 3 as the size and the complexity of the data structure grow with its depth. The sublinear decrease in runtime can be observed as the number of processors are doubled for each benchmark. Datasets with small amounts of features, i.e. compas-binary, reach their maximum parallel speedup for a less amount of processors in comparison with datasets with great amounts of features, i.e. hypothyroid and audiology. This was obviously expected beforehand, because more features means that more computations need to be done per thread and less features means that more threads may get idle while executing the program.

Figure 6 covers the average optimal tree experiment discussed in section 5.1.2. In 10 runs, single experiments are performed with a fixed ratio between the train and the test data (70%-30%). Altogether, the average train and the average test accuracy are plotted as well as the average parallel execution times T_p for $depth \in [3, 4]$.

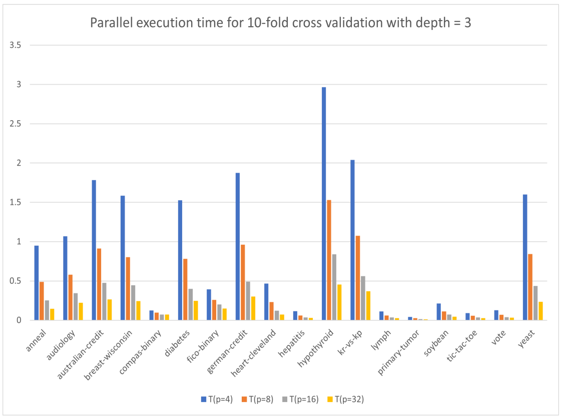
As expected, the train accuracy is greater than the test accuracy for every benchmark, since there are greater amounts of misclassifications when evaluating the model on unseen data. Compared to 10-fold cross validation, even though this experiment uses more records in the test set, the overall test accuracy is approximately the same for both experiments. In terms of runtime, the total duration of this experiment is slightly shorter than the total duration of 10-fold cross validation, because each run constructs decision trees using less train data (70%) for this experiment compared to the train set of the previous experiment that consists of data from 9 out of 10 folds (90%).



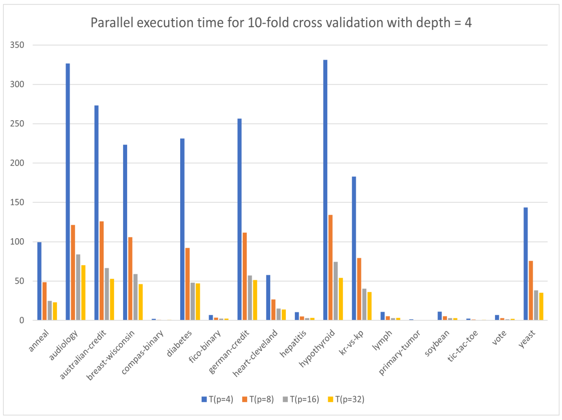
(a) Mean of test accuracy with $depth \in [3, 4]$



(b) SD of test accuracy with $depth \in [3, 4]$

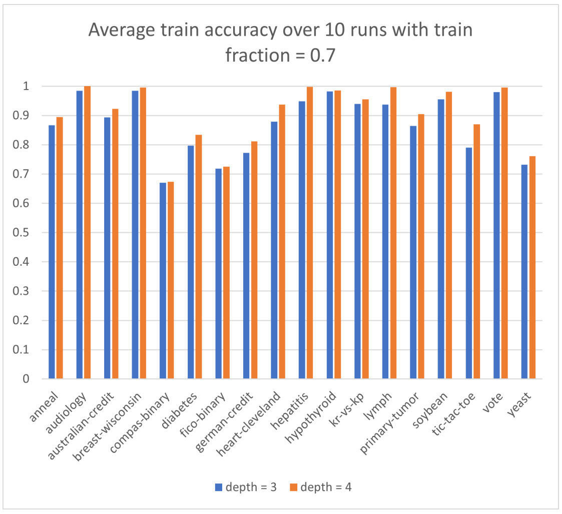


(c) Aggregated T_p in seconds with $depth = 3$

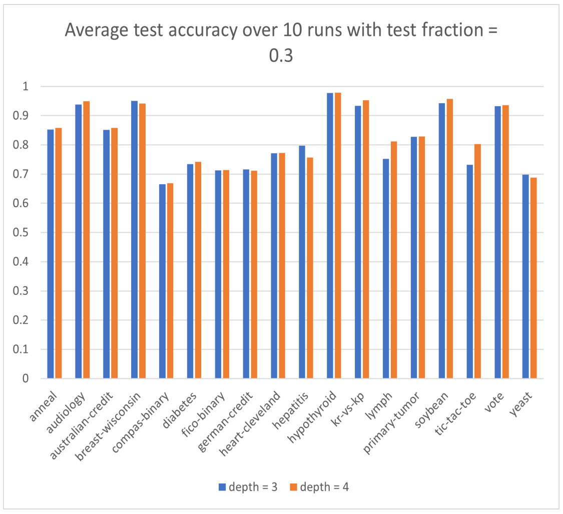


(d) Aggregated T_p in seconds with $depth = 4$

Figure 5: 10-fold validation on binarised datasets to construct and evaluate optimal decision trees of $depth \in [3, 4]$.

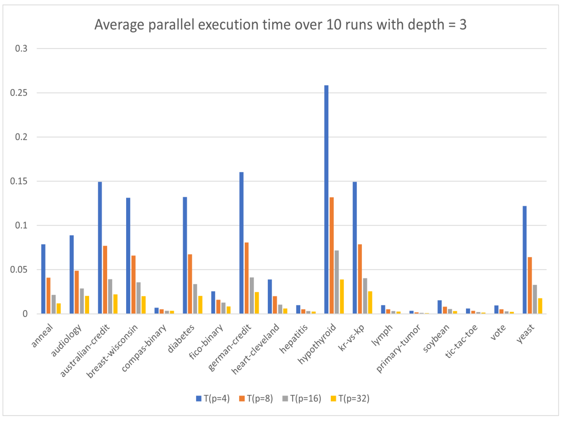


(a) Mean of train accuracy with $depth \in [3, 4]$

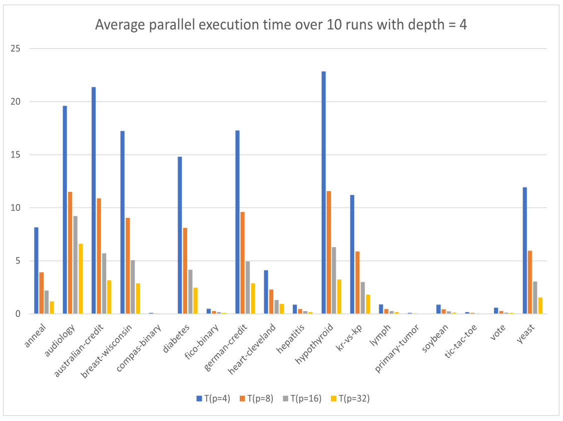


(b) Mean of test accuracy with $depth \in [3, 4]$

Figure 6: Average optimal tree experiment of 10 runs on binarised datasets to train and evaluate optimal decision trees of $depth \in [3, 4]$. For each run, 70% of the dataset is used as the train set and 30% of the dataset is used as the test set.



(c) Mean of T_p in seconds with $depth = 3$



(d) Mean of T_p in seconds with $depth = 4$

Figure 6: Average optimal tree experiment of 10 runs on binarised datasets to train and evaluate optimal decision trees of $depth \in [3, 4]$. For each run, 70% of the dataset is used as the train set and 30% of the dataset is used as the test set. (cont.)

6 Responsible Research

Every study requires to have some discussion on its ethical aspects. The research is mainly focused on the ethical implications of optimal decision trees. The program reads discrete binarised data from dataset files in order to model the decision tree as a non-linear classifier [1]. It is important to note that the data is fixed but not continuous, meaning that every record is already present and is not fetched from some real-time streaming source. The datasets used for this research are publicly available [26]. However, some ethical review is still significant to find out more about sharing the work and its possible consequences.

One of the main problems that can be encountered is that machine learning researchers may experience intrinsic loss of freedom. Contributors are sometimes not able reveal their work, because they fail to set limits to their privacy, which violates a basic human right [27]. Developers' self-ownership and self-growth are also harmed when they lose their freedom [28, p. 37]. Therefore, researchers should always be capable to decide whether they want their work of optimal decision trees to be published on external platforms.

Another issue is that research-related information can become accessible to third parties. Disclosing belongings to parties outside the organisation of the research team can have the severe consequence that the optimal decision tree algorithms are used in unrelated research fields without getting consent from the actual contributors. The information may also be further spread to other organisations once a third party loses interest in the research. Therefore, it is always necessary to determine with whom the components of this research are shared.

In case there are researchers who would prefer to make the parallel optimal decision tree model reproducible for their work, they would have to abstract the constituent components that build up the research. These components include the binarised datasets that are used for building optimal decision trees, the algorithms of the MurTree paper [6] to construct optimal decision trees, and lastly, the mentioned OpenMP extensions which are necessary to perform parallelization on optimal decision trees. Once these components are abstracted, it becomes possible to establish reproducibility of this research for further work.

7 Conclusions and Future work

This paper presented how to apply parallelism on the MurTree algorithm using multiple processors. The algorithm produces decision trees that represent the data in the best possible way, i.e., tree representations with the minimal misclassification score on their root. The experimental study shows the improvement of the parallel program for different numbers of processors on different benchmarks and analyzes performance metrics such as the parallel execution time, the speedup and the efficiency of the program.

To come back to the aims set for this study, each research question has been answered from a sufficient to a high degree. The trade-off has been investigated regarding the performance of the program. As the number of CPUs increases, the speedup of the program increases as well while the efficiency of the program decreases. The parallel directives of OpenMP partition the computations in chunks of blocks and schedule each block to some available thread. Since the algorithm performs multiple computations for a particular feature, i.e., looping over the different possible tree sizes, the problem size is also distributed among the threads. Therefore, the parallelization strategy deals with both data and task parallelism of the program.

With regard to the third research question, the parallel implementation carefully considers different OpenMP clauses to avoid having problems like a program with a data race or a program where some threads remain idle. In order to prevent multiple threads from simultaneously updating a shared variable, such as the best decision tree, the reduction clause specifies to the program that such a variable should not be accessed by multiple threads at the same time. Additionally, the collapse clause is used to expand the total number of iterations for the general depth algorithm and guarantees a better load balance over the available threads.

There are some important limitations which have not been considered for this study. The optimal decision trees are not computed using an incremental approach. Also, the algorithm has not been optimized with the similarity-based lower bounding approach discussed in the MurTree paper. Aside from that, there are still some open issues for exploiting multi-core parallelism on optimal decision trees. The OpenMP paradigm schedules threads using the concept of work sharing, however, real world applications may also require other scheduling approaches, such as the topic of work stealing which was not emphasized for this study. Moreover, multiple threads may still have the performance issue of false sharing, which is still an open research field for this study.

There are still some recommended features that can be added to this research with regard to parallelizing optimal decision trees. One such feature is to parallelize the MurTree algorithm using OpenMPI. In addition, a study can be made on the communication overhead caused by the processes that each go over a different part of the program. Other possible setbacks should then also be analyzed such as deadlocks between multiple processes. To conclude, if the aforementioned recommendations are implemented, then it would be possible to make comparisons between more than one parallel approach on different benchmarks based on the performance metrics provided in this study.

References

[1] M. Kuhn and K. Johnson, “Nonlinear classification models,” in *Applied Predictive Modeling*. New York, NY: Springer New York, 2013, pp. 329–367, ISBN: 978-1-4614-6849-3. DOI: 10.1007/978-1-4614-6849-3_13. [Online]. Available: https://doi.org/10.1007/978-1-4614-6849-3_13.

[2] *Decision tree algorithm explained with examples*, Great Learning, Feb. 2020. [Online]. Available: https://www.mygreatlearning.com/blog/decision-tree-algorithm/.

[3] L. Hyafil and R. L. Rivest, “Constructing optimal binary decision trees is np-complete,” *Information Processing Letters*, vol. 5, no. 1, pp. 15–17, 1976, ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(76)90095-8. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0020019076900958.

[4] L. Breiman, J. Friedman, C. Stone, and R. Olshen, *Classification and Regression Trees*. Taylor & Francis, 1984, ISBN: 9780412048418. [Online]. Available: https://books.google.nl/books?id=JwQx-WOmSyQC.

[5] D. Bertsimas and J. Dunn, “Optimal classification trees,” *Machine Learning*, vol. 106, Jul. 2017. DOI: 10.1007/s10994-017-5633-9.

[6] E. Demirovi’c, A. Lukina, E. Hébrard, J. Chan, J. Bailey, C. Leckie, K. Ramamohanarao, and P. J. Stuckey, “Murtree: Optimal classification trees via dynamic programming and search,” *ArXiv*, vol. abs/2007.12652, 2020.

[7] Q. Meng, G. Ke, T. Wang, W. Chen, Q. Ye, Z. Ma, and T.-Y. Liu, “A communication-efficient parallel algorithm for decision tree,” *CoRR*, vol. abs/1611.01276, 2016. arXiv: 1611.01276. [Online]. Available: http://arxiv.org/abs/1611.01276.

[8] D. Khaldi, P. Jouvelot, C. Ancourt, and F. Irigoin, “Task parallelism and data distribution: An overview of explicit parallel programming languages,” vol. 7760, Sep. 2012. DOI: 10.1007/978-3-642-37658-0_12.

[9] T. Rauber and R. Gudula, *Parallel Programming: for Multicore and Cluster Systems*. Springer, 2013.

[10] B. Van Houdt, “Randomized work stealing versus sharing in large-scale systems with non-exponential job sizes,” *IEEE/ACM Transactions on Networking*, vol. 27, no. 5, pp. 2137–2149, 2019. DOI: 10.1109/TNET.2019.2939040.

[11] T. Mattson, “An introduction to openmp,” Feb. 2001, pp. 3–3, ISBN: 0-7695-1010-8. DOI: 10.1109/CCGRID.2001.923161.

[12] *Openmp architecture*, “hpc-wiki.info”, 2021. [Online]. Available: https://hpc-wiki.info/hpc/OpenMP.

[13] *Core elements*, “Wikiwand.com”, 2021. [Online]. Available: https://www.wikiwand.com/en/OpenMP.

[14] R. Netzer and B. Miller, “What are race conditions? - some issues and formalizations,” *ACM letters on programming languages and systems*, vol. 1, Sep. 1992. DOI: 10.1145/130616.130623.

[15] W. Bolosky and M. Scott, “False sharing and its effect on shared memory,” *SEDMS IV*, Aug. 1993.

[16] R. Graham, T. Woodall, and J. Squyres, “Open mpi: A flexible high performance mpi,” Sep. 2005, pp. 228–239. DOI: 10.1007/11752578_29.

[17] M. Singhal, “Deadlock detection in distributed systems,” *Computer*, vol. 22, pp. 37–48, Dec. 1989. DOI: 10.1109/2.43525.

[18] *Perfect binary tree*, “Programiz.com”, 2021. [Online]. Available: https://www.programiz.com/dsa/perfect-binary-tree.

[19] V. Balabanov, M. Kaufman, D. Knill, D. Haim, O. Golovidov, A. Giunta, R. Haftka, and B. Grossman, “Dependence of optimal structural weight on aerodynamic shape for a high speed civil transport,” *6th Symposium on Multidisciplinary Analysis and Optimization*, Dec. 1996. DOI: 10.2514/6.1996-4046.

[20] J. N. Amaral, *Why high performance computing?* Apr. 2002. [Online]. Available: https://webdocs.cs.ualberta.ca/~amaral/talks/MACI-Apr2002/sld080.htm.

[21] S. Feki and M. Smaoui, “Collapse clause - an overview | sciencedirect topics,” Jan. 2017. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/collapse-clause.

[22] “Reduction clause - an overview | sciencedirect topics,” 2021. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/reduction-clause.

[23] *Pragma omp declare reduction*, 2016. [Online]. Available: https://www.ibm.com/docs/en/xl-c-and-cpp-linux/16.1.1?topic=parallelization-pragma-omp-declare-reduction.

[24] D. Berrar, “Cross-validation,” in. Jan. 2018, ISBN: 9780128096338. DOI: 10.1016/B978-0-12-809633-8.20349-X.

[25] D. Helmbold and C. Mcdowell, “Modeling speedup (n) greater than n,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 1, pp. 250–256, May 1990. DOI: 10.1109/71.80148.

[26] @UCLouvain, *Datasets of pydl8.5*, https://github.com/aia-uclouvain/pydl8.5/tree/master/datasets, 2019.

[27] *Universal declaration of human rights.* [Online]. Available: https://www.un.org/en/universal-declaration-human-rights/index.html (visited on Jun. 18, 2020).

[28] J. H. Reiman, “Driving to the panopticon: A philosophicalexploration of the risks to privacy posed by thehighway technology of the future,” *Santa Clara High Technology Law Journal*, vol. 11, no. 1, 1995.