Future Generation Computer Systems 158 (2024) 308–322

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Parallel approaches for a decision tree-based explainability algorithm

Daniela Loreti *, Giorgio Visani

Department of Computer Science and Engineering, University of Bologna, Viale del Risorgimento 2, Bologna, 40136, Italy

ARTICLE INFO

Keywords:
Parallel computing
Global and Local Explainability
Parallel decision tree building

ABSTRACT

While nowadays Machine Learning (ML) algorithms have achieved impressive prediction accuracy in various fields, their ability to provide an explanation for the output remains an issue. The explainability research field is precisely devoted to investigating techniques able to give an interpretation of ML algorithms' predictions. Among the various approaches to explainability, we focus on GLEAMS: a decision tree-based solution that has proven to be rather promising under various perspectives, but suffers a sensible increase in the execution time as the problem size grows.

In this work, we analyse the state-of-the-art parallel approaches to decision tree-building algorithms and we adapt them to the peculiar characteristics of GLEAMS. Relying on an increasingly popular distributed computing engine called Ray, we propose and implement different parallelization strategies for GLEAMS. An extensive evaluation highlights the benefits and limitations of each strategy and compares the performance with other existing explainability algorithms.

1. Introduction

As ML techniques have seen a boost in interest during the last decade, a key research field strictly related to ML has taken shape in the direction of explainable Artificial Intelligence (AI). Indeed, ML algorithms have achieved impressive prediction accuracy in various fields – spanning from image processing and sentiment analysis, to recommendation systems and tabular data – but the intrinsical structure of most of the algorithms is such that it is impossible to provide an explanation for the output. They lack the crucial ability to clarify the rationale for that prediction. In some fields, this limitation represents a major downside that renders almost useless the application of AI techniques [1].

For this reason, different studies have been conducted in order to provide general methods to explain the prediction of a ML model [2–6]. Explainability methods are generally classified according to their ability to provide local or global explanations, i.e., respectively, the ability to describe the reasons for the prediction of a specific input point; or the ability to explain the ML model as a whole. In particular, GLEAMS (Global & Local ExplainAbility of black-box Models through Space partitioning) [6] is a promising approach that relies on the construction of a decision tree with peculiar characteristics to provide an accurate description of the ML model. One of the advantages of GLEAMS with respect to other existing methods is its ability to naturally provide both local and global explanations: a key feature that other explainability algorithms can achieve at the cost of additional computations.

Nonetheless, the execution time of GLEAMS is heavily influenced by the dimension of the problem. In its more recent implementation [6, Ch. 8], GLEAMS is an intrinsically sequential algorithm: it cannot take advantage of a multicore machine, let alone a network of computing nodes. In this work, we argue that resorting to a parallel approach might be crucial, not only to improve the performance in terms of execution time but also to reduce the overall power consumption of the algorithm—a matter of major concern for modern data centres.

Albeit there exists a large literature on how to parallelize decision tree-building algorithms, GLEAMS' peculiar management of the input training set makes the latest and most promising approaches scarcely applicable. In this work, we focus on how GLEAMS can be modified to take advantage of a multiprocessor system or a network of computing nodes in order to boost both its time and power consumption performance. For this purpose, we rely on an increasingly popular distributed computing engine, Ray [7], known to be particularly suitable for the parallel execution of ML tasks. We evaluate different possible implementations of parallel approaches through extensive scalability tests, thus highlighting the advantages and shortcomings of each implementation. Finally, since energy consumption is becoming more and more important for large data centres, we investigate the advantages that a parallel approach can bring on this front.

The main contributions of this work are the following.

- A study of the parallelization opportunities for GLEAMS conducted after a detailed investigation of the state-of-the-art techniques for parallel decision tree-building algorithms.

* Corresponding author.
E-mail address: daniela.loreti@unibo.it (D. Loreti).

https://doi.org/10.1016/j.future.2024.04.044
Received 29 September 2023; Received in revised form 5 April 2024; Accepted 22 April 2024
Available online 30 April 2024

- Four novel parallel approaches to the parallelization of GLEAMS, which take inspiration from the related work in this field and make use of Ray distributed processing engine.
- An extensive performance comparison of the proposed approaches to empirically identify the strengths and weaknesses of each, the power consumption reduction that can be achieved through parallelization, and a comparison with state-of-the-art explainability techniques in terms of time to compute the solution.

Regarding the latter contribution, we clarify that the accuracy of GLEAMS as an explainability method has been already studied in previous works [6,8,9]. All the parallelization techniques presented in this work do not influence the algorithm's ability to provide an explanation for a ML model but operate only to subdivide the computational work over different nodes in order to obtain a performance improvement.

The article is structured as follows. Section 2 provides a brief overview of existing explainability techniques and introduces GLEAMS algorithm in detail. Section 3 reports a literature review of state-of-the-art approaches to the parallelization of decision tree-building algorithms. Section 4 introduces Ray's programming paradigm and describes how it can be employed to implement the proposed parallelization strategies for GLEAMS. Section 5 presents the empirical evaluation and Section 6 draws conclusions.

2. Background on GLEAMS

The GLEAMS [6,10] technique aims to offer an explanation – in the form of a decision tree T – for a generic black-box ML model f. It can be applied to both regression and classification scenarios.

2.1. Related work on explainability

The idea of approximating globally a complex ML model f by a simpler *surrogate model* relying on a tree-based partition of the space can be dated back at Craven and Shavlik [5]. The proposed method, TREPAN, approximates the outputs of a neural network by a decision tree, choosing the splits using the *gain ratio* criterion [11]. More recently, inspired by Gibbons et al. [12], Zhou and Hooker [13] proposed to use another split criterion, based upon an asymptotic expansion of the Gini criterion. Both these approaches are limited to the classification setting and also require access to the training data (which is implicitly assumed to have a good covering of the input space).

From the interpretability side, a standard way to explain f, which is closely related to our approach, is to compute *attributions* scores, i.e. a measure of importance, for each variable in the dataset. Lei et al. [14] exclude a specific feature from f and define its impact as the deterioration of model predictions. On the contrary, Partial Dependence Plots (PDP) developed by Friedman [15] measure the sensitivity of f to changes in the specific feature, isolating its effect from the ones of the other features.

Particularly relevant for our work are the papers [3,4]. Ribeiro et al. [4] propose LIME, a well-known local, model-agnostic explanation algorithm. The method creates a surrogate linear model valid in the local region around the individual to be explained. Specifically, LIME generates random points all over the $\mathbb{R}^d$ space of the variables used to train the ML model, then smoothly assigns higher weights to points closer to the reference, effectively considering only points in a small neighbourhood. The weights' decay is governed by the kernel width parameter. The generated points and respective weights are used to train a linear local model, whose coefficients are regarded as LIME explanations.

LIME has already been employed several times in medicine, such as on Intensive Care data [16] and cancer data [17,18]. However, the technique is known to have few important weak points. The most notable ones are instability [19] (i.e., repeated LIME explanations on the same individual with equal settings may achieve quite different results) and locality (i.e., to determine the proper size of the neighbourhood for the local explanation). The former hindrance is mainly caused by the randomness introduced in the sampling step and has been addressed, among others, in [8] where Stability Indices inform the user of possible faults and help her improve the explanations. On the other hand, the explanation's locality is extremely important, but the kernel width parameter lacks a straightforward interpretation and prescriptions on how to tune it. In recent years, the work [9] provided an automated optimization policy to select the best kernel width value. Nonetheless, it is still hard to know the precise neighbourhood boundaries where the explanation is valid. From a regulatory point of view, such a desideratum is much needed to trust the explanations. A change of paradigm was necessary, towards a global explanation method exploiting the same LIME linear explanations but providing precise and clear prescriptions to be useful in practice. GLEAMS [6] fills the gap, by producing a set of linear models covering the entire variable space, each one valid in a precisely delimited region. GLEAMS effectively extends the LIME intuition to the realm of global explanations, while addressing an important regulatory requirement.

Similarly to LIME, SHAP [3] introduces the idea of decomposing f predictions into single variables *attributions*. The technique samples combinations of features S and average changes in single instance prediction between the restricted $f(S \cup j)$ against $f(S)$, obtaining local *attributions* for the feature j. Global SHAP *attributions* arise by averaging local attributions over the entire dataset.

Attribution methods usually have to be recomputed for each new example, a caveat that GLEAMS proposes to overcome by computing a global surrogate and relying on it to provide explanations for any new unseen data point. GLEAMS is not the first algorithm to provide explanations on a global scale: Ribeiro et al. [20] propose Anchors, a method extracting local subsets of the features (anchors) that are sufficient to recover the same prediction while having a good global coverage. Setzu et al. [21] propose to *aggregate* local explanations: starting from local decision rules, GlocalX combines them in a hierarchical manner to form a global explanation of the model. In the *ad hoc* setting, Harder et al. [22] propose to train an interpretable and differentially private model by using several local linear maps per class: the pseudo-probability of belonging to a given class is the softmax of a mixture of linear models. This approach is limited to the classification setting, as with GlocalX and Anchors.

In summary, LIME and SHAP are the most widely utilized explainability methods due to their user-friendly nature, robust mathematical foundations, and abundant online resources, along with clear and comprehensive code implementations. Consequently, in the upcoming experimental section, we will conduct a computational comparison of GLEAMS with LIME and SHAP, as these two methods are the primary contenders within the field of explainability.

2.2. GLEAMS sequential implementation

Given a set $X = \{x_1, \dots, x_i, \dots, x_N\}$ of d-dimension input points (i.e, $x_i \in \mathbb{R}^d$), the ML model f computes a prediction for each point in X. We denote with Y the set of output target values (i.e., $Y = \{y_i = f(x_i)\ \forall i = 1, \dots, N\}$). Each y_i can be either a continuous variable predicted by the model (in the case of regression), or the probability of a sample x_i being predicted as class c out of C possible classes (in case of classification).

GLEAMS operates to build a piecewise linear approximation of f as depicted in Fig. 1. In very simple terms, it aims to split the input space into non-overlapping parts (similar to what a common decision tree-building algorithm would do) and associate to each part a local linear approximation of f. The result is therefore a peculiar decision tree T, where each leaf contains the linear approximation of a portion of the input space. The set of coefficients obtained from these linear approximations is regarded as proxies of the importance that each feature has on the whole input space, thus providing a *global explanation* for f.

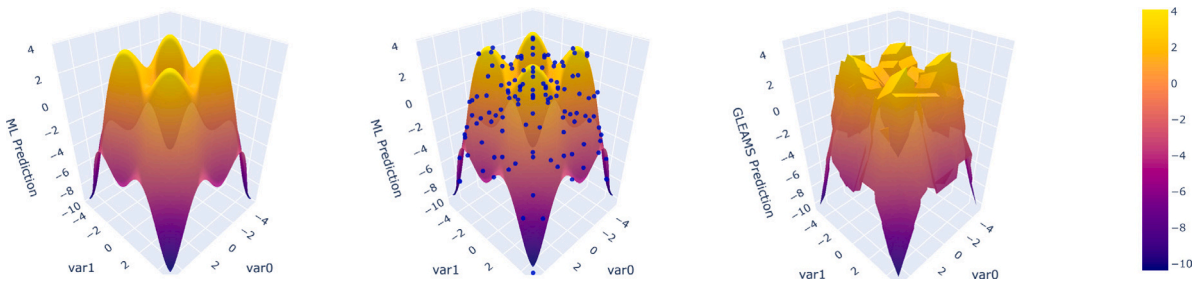

Fig. 1. Overview of the GLEAMS explanation tree construction. *Left panel:* the black-box f model maps the input space (here $[0,1]^2$) to $\mathbb{R}$, which we can visualize as a surface. *Middle panel:* we generate N Sobol points, giving rise to N measurement points on the surface (in blue). *Right panel:* we fit the GLEAMS piecewise-linear global surrogate model on the measurement points by recursively splitting the input space, in a tree fashion [6].

It is important to underline that, given its characteristics, GLEAMS can also provide *local explanations* of predictions for given unseen examples. Any new example will fall into a leaf of T and the coefficients of the linear approximation of that leaf will help to understand which of the model features influenced the prediction the most.

As shown in Algorithm 1,[1] GLEAMS starts by synthetically generating a set of points (line 2) in the input space. To get good coverage of the input space, the generation is not pseudorandom but follows the criteria of Sobol [23]. For each Sobol point x_i, the algorithm computes the prediction y_i using the input ML model f (line 3). Finally, the RecTree procedure is called to actually compute the tree by means of the Model-Based recursive partitioning (MOB) algorithm [24].

Algorithm 1 Computation of the global surrogate model used by GLEAMS.

Input: f: black-box model ; $\mathcal{X}$: boundaries of the input space
Output: T: decision tree explanation for f

1: **procedure** GlobalSurrogate($f, \mathcal{X}$)
2: generate $X = \{x_1, \dots, x_N\}$ Sobol points in $\mathcal{X}$
3: $y_i \leftarrow f(x_i)\ \forall i \in \{1, \dots, N\}$
4: $Y \leftarrow \{y_1, \dots, y_N\}$
5: **return** $T \leftarrow$ RecTreeX, Y

The pseudocode of RecTree is shown in Algorithm 2. It takes as input the set of (generated) points X and their predictions Y computed through f, and recursively operates to emit T, the root node of the decision tree explanation of f. Like in any tree-building algorithm, starting from the root node, a procedure is called (GetBestSplit in line 7) to compute the point in which it is best to split the input space. The best-split point is identified here by a (j, t_j) pair, where j is the feature and t_j is the value of the split on j (more details on this procedure are given in Algorithm 3). According to the computed best-split point, the input set X is divided into two parts X^ℓ and X^r (line 9), and the RecTree procedure is recursively called on each of them to generate the left and right child nodes. In the serial implementation, the decision tree is therefore constructed following a left-most, depth-first traversal.

The recursion terminates when, for a given node of the tree, the linear regression computed on X, Y (procedure LinReg in line 5) is deemed good enough, i.e. it is a good linear approximation of f in the subspace of the current node, in terms of R^2 metric. In such a case, the algorithm returns a leaf node (line 6).

During the algorithm execution, it may also happen that, for a given candidate leaf, the number of points in X is not enough to correctly assess if the regression displays a good fit.[2] In this case, we need to generate new points in the current node subspace. To do so, we simply call again Algorithm 1 (lines 2 to 4).

Algorithm 2 Recursive construction of T.

Input: X: set of input points pertaining to the current node; Y: values of f on X; n_{min}: minimum size of each leaf.
Output: T: root note of the decision tree that approximates f

1: **procedure** RecTree(X, Y)
2: **if** $|X| < 2n_{min}$ **then**
3: $\mathcal{X} \leftarrow$ boundaries of the input space of X
4: **return** GlobalSurrogate$f, \mathcal{X}$
5: $\hat{f} \leftarrow$ LinRegX, Y
6: **if** $R^2(X, \hat{f}) > \rho$ **then return** T as leaf
7: $(j, t_j) \leftarrow$ GetBestSplit$X, Y, \hat{f}$
8: $T \leftarrow$ NodeX, Y, j, t_j
9: $(X^\ell, Y^\ell, X^r, Y^r) \leftarrow$ SplitDataX, Y, j, t_j
10: $T.\ell \leftarrow$ RecTreeX^ℓ, Y^ℓ
11: $T.r \leftarrow$ RecTreeX^r, Y^r
12: **return** T

Eventually, Algorithm 3 takes as input X, Y and the local linear approximation $\hat{f}$ in the form of a linear regression formula, to compute the best-split point for the current node. $\hat{f}$ is used on X to estimate $\hat{Y}$, i.e. the linear approximation of target values. The GetScores procedure uses it to compute the array of scores $\mathcal{S}$. Intuitively, GetScores compares the similarity between Y and $\hat{Y}$, per each point in X. Such array is then used to produce a cumulative score process $B^{(j)}$ per feature, containing information about the goodness of any possible split on the generic variable j. From $B^{(j)}$ we compute the objective function m_j – which we want to maximize – and choose the best variable for the split as the one yielding maximum m_j value. For the mathematical details of this procedure, we refer to the work [6].

From a parallelization point of view, the GetScores procedure to compute the array $\mathcal{S}$ is the most computationally intensive part and it requires the whole X dataset. Contrarily to other tree algorithms, we cannot decompose its computation into different tasks, each one taking care of a specific feature. Indeed, in order to compute the score of any dimension j, the values of the input examples for j are not sufficient: the procedure needs the values of the input points for *all* the features.

For completeness, Table 1 reports the performance of GLEAMS on three datasets of real-world data w.r.t. other methods when explaining two types of ML models: XGBoost [25] and multi-layer perceptron (MLP). A detailed description of these experiments is reported in [6]. The comparison is carried out in terms of local and global monotonicity, and recall. Briefly, local monotonicity reports how well the algorithm's attributions reflect the variations in prediction locally (i.e., in each data

[1] The pseudocodes in this section are taken from [6] but many mathematical foundations are omitted for space reasons. Please refer to [6] for any further details.

[2] Intuitively, for example, computing linear regression on 3 points in a 3-dimensional space would yield a plane perfectly fitting the points. However, this is just due to the limited number of input points while it is not ensured that the plane truly approximates f.

Table 1
Performance of GLEAMS w.r.t. state-of-the-arts.
Source: Taken from [6].

Dataset	Metrics	LIME		SHAP		PDP		GLEAMS	
		XGB	MLP	XGB	MLP	XGB	MLP	XGB	MLP
Wine[a]	local m.	0.38	0.51	0.40	0.53	0.56	0.84	0.51	0.77
	global m.	0.24	0.74	0.27	0.78	0.70	0.89	0.36	0.85
	recall	0.89	0.80	1.0	0.87	0.50	0.50	0.77	0.75
House[b]	local m.	0.05	0.63	0.52	0.82	0.51	0.83	0.37	−0.19
	global m.	0.23	0.63	0.37	0.81	0.38	0.82	0.47	0.24
	recall	0.85	0.74	0.99	0.69	0.40	0.40	0.61	0.58
Park.[c]	local m.	0.10	0.26	0.47	0.55	0.28	0.49	0.50	0.01
	global m.	0.31	0.46	0.41	0.71	0.52	0.66	0.30	0.36
	recall	0.86	0.77	0.97	0.80	0.40	0.40	0.50	0.60

[a] https://archive.ics.uci.edu/ml/datasets/wine+quality.
[b] https://www.openml.org/search?type=data&status=active&id=42092.
[c] https://archive.ics.uci.edu/ml/datasets/parkinsons+telemonitoring

split); global monotonicity measures how well the attributions reflect the variations in prediction on the whole input space along a certain feature; and recall reports the average recall over all points of the test set. Refer to [6] (Chapter 3.3 and 4.3) for a detailed explanation of these metrics and datasets. In Table 1, GLEAMS' runs on 2^{15} Sobol points are compared with LIME for tabular data [4], SHAP [3] and PDP [15] using default parameters. In general, GLEAMS is on par with state-of-the-art methods in terms of the reliability of explanations. Indeed, it provides better local and global monotonicity w.r.t. LIME and SHAP on the *Wine* dataset, while (as expected) its performance starts to degrade when the number of features increases, as in the *House* and *Parkinson* datasets. This behaviour can be partially mitigated by increasing the number of Sobol points. As for the recall metric, GLEAMS performance is slightly worse than SHAP, but comparable with LIME, and systematically better than PDP.

Algorithm 3 Get the best split for a given node.

Input: X: set of Sobol points in the current node; Y: values of f on X; d: number of features in the input, $\hat{f}$: linear regression formula approximating f on the given node input space.
Output: j^*: feature to split; t_{j^*} value of the split

procedure GetBestSplit($X, Y, \hat{f}$)
  $\hat{Y} \leftarrow \hat{f}(X)$
  $S \leftarrow$ GetScores $\hat{f}, \hat{Y}, X$
  for $j = 1$ to d **do**
    (m_j, t_j)=GetBestSplitForFeature X, Y, S, j
  $j^* \leftarrow \text{argmax}_{j=1,\dots,d}\, m_j$
  t_{j^*}, value of the split
  return (j^*, t_{j^*})
procedure GetBestSplitForFeature(X, Y, S, j)
  sort $[X, Y, S]$ according to feature j
  $B^{(j)} \leftarrow$ CumulativeScoreProcess S
  $m_j \leftarrow \max_{i=1,\dots,N} \left\| B^{(j)}(i) \right\|_1$
  $t_j \leftarrow \text{argmax}_{i=1,\dots,N} \left\| B^{(j)}(i) \right\|_1$
  return m_j, t_j

3. Literature review on parallel decision tree learning algorithms

GLEAMS' objective is constructing a decision tree to explain an input ML model. When it comes to parallelizing the training phase of a decision tree, extensive literature exists: several approaches have been proposed in the last decades [26]. The work by Amado et al. [27] distinguishes between three categories of parallel decision tree-building algorithms according to their choice of domain decomposition: *task parallelism, horizontal data parallelism and vertical data parallelism*. The first class of works (*task parallel* algorithms) stems from the intuition that different subtrees can be computed independently. For this reason, the works falling into this category propose a decomposition of the domain such that each processor is in charge of a tree's node (or a subtree). The main observed drawback of this strategy is the difficulty of balancing the resulting computational load to be assigned to different machines. Indeed, as it is not possible to foresee the structure of the final tree, it is also not possible to understand in advance what is the best number of parallel processors to employ: if a tree is particularly unbalanced in its structure, so will the parallel execution. Furthermore, if the final tree is very deep and not particularly wide, this can limit the degree of parallelism that can be reached. Therefore, a second set of works has been proposed, which Amado et al. [27] refer to as *data parallel* algorithms. Instead of parallelizing the structure of the tree, they try to subdivide the computation by splitting the input training set. As this operation can be performed in two ways, Amado et al. identify the category of *vertical* and *horizontal data parallelism*. Vertical data parallelism refers to the possibility of splitting the training set by variable: each different computing processor receives only the vector of values referring to a particular variable for the whole input examples, and the vector of the target. Horizontal parallelism instead refers to the possibility of providing each processor with a partition of the input examples, i.e., the whole set of variables (and the target) for just a subset of the examples in the training set is given to each computing node.

The work [28] offers examples of all three parallel approaches applied to C4.5 [29]: Dynamic Task Distribution (DTD) implementing task parallelism with a master processor that allocates a subtree of the decision tree to any idle slave processor; the DP-rec scheme distributes the data horizontally and builds decision tree one node at a time; while the DP-att scheme distributes the attributes, bringing the advantages of being both load-balanced and requiring minimal communications. However, DP-att suffers from limited scalability.

Pearson [30] proposes a particular example of vertical data partitioning in which this strategy is coupled with a master-worker infrastructure in charge of the assignment of the subtasks to computing processors. Since Pearson's solution dates back to 1994, it could not take advantage of modern distributed computing engines (like Spark, Hadoop or Ray). Hence, when the degree of parallelism is too high, the algorithm's computation time is dominated by the overhead of task assignment and bookkeeping activities on the master.

Since many decision tree training algorithms rely on large datasets, which cannot fit a single machine RAM, *horizontal data parallelism* is the most explored category. One of the first attempts to resort to horizontal data parallelism dates back to 1995's work by Kuffrin [31], which proposes a "systolic shift" phase to put together the results of the computations carried on separately by each processor. Albeit effective in finding the best-split point, this strategy entails a relevant number of messages exchanged, which can become a significant bottleneck. Other well-known horizontal data parallel approaches are SPRINT [32]

and SLIQ [33], where the authors propose the employment of additional data structures to avoid sorting the same array multiple times. Nonetheless, the problem of the significant number of exchanged messages remains unsolved, because for all these algorithms an all-to-all broadcast is needed each time the identification of a global best-split point is required. ScalParC [34] enhances SPRINT with a distributed hash table that reduces the latency of the splitting decision, especially for continuous attributes. The works [35,36] propose to exchange quantized histograms between machines when estimating the global attribute distributions. However, in all these works, the communication cost is still proportional to the total number of attributes.

An actual reduction of the communications is proposed in [37], where each computing processor builds its own decision tree based on a subset of the training examples. An unseen example is evaluated by all the models and a majority vote is taken to determine its class. Voting is also employed in [38–41]. In particular, Meng et al. [40] show through theoretical analysis that the proposed PV-Tree algorithm can learn a near-optimal decision tree because it can find the best attribute with a large probability. A similarity-based approach is instead employed in the ensamble technique of [42] to select the tree that is most similar to others as the best representative of the entire dataset.

Amado et al.'s classification of the works into approaches that parallelize the model or the input data (in different ways) is indeed common to other ML-related fields [43–47]. Alternative to Amado et al.'s classification is instead the subdivision of parallel decision tree algorithms by Srivastava et al. [48], which propose to distinguish between *synchronous* and *partitioned tree construction* depending on the fact that the final tree is either, respectively, built step by step by all the processors (which entails a global synchronization at each iteration, like in most horizontal data parallel approaches) or subdivided into pieces independently computed by different processors (like in pure task parallelism or in approaches that combine task and vertical data parallelism). For better clarity, in this work, we have decided to employ the classification of Amado et al. [27].

It is important to underline that, in more recent years, with the increasing interest and diffusion of the MapReduce approach [49], several works [50–54] have explored the possibilities to apply this programming paradigm to the training of large decision trees with horizontal data parallelism. Panda et al. [53] in particular, presented PLANET, a scalable distributed framework that applies MapReduce to distribute and scale decision tree induction to very large datasets. Despite the employment of MapReduce, the communication overhead of the algorithm is significant. Dai et al. [52] propose a parallelization of the classical C4.5 algorithm which relies on new data structures (the attribute, count, and hash tables) to minimize the communication cost. An initial MapReduce step is employed to compute these data structures, then other MapReduce steps build the tree on the basis of the structures. MapReduce-based parallelization has been applied to fuzzy decision trees too [55,56].

All these works propose a Reduce step in which results computed independently by different processors are composed together. This strategy allows for a decrease in the number of messages exchanged, but the key point of weakness stands precisely in the way the partial results are combined.

Recent years have also seen a rise in the employment of GP-GPUs for the acceleration of decision tree induction. Task and vertical data parallelisms are combined in the works [57,58], while Jurczuk et al. [59–63] focus on GPU-based evolutionary algorithms (EAs) using horizontal data parallel approaches. The technique of [59] is enhanced in [64] to take advantage of multiple GPUs. Finally, CRO-DT [65] uses a matrix encoding of the decision tree and a GPU implementation to speedup an ensemble algorithm and evolve better univariate trees.

Table 2 summarizes the main characteristics of the papers related to our work. For each solution, it highlights the chosen domain decomposition: task (T), horizontal (HD) or vertical (VD) parallelism [27], the strategy adopted for the tree construction: synchronous or partitioned [48], the implementation framework employed, the decision tree type on which it is tailored, and the distributed platform (if any) that supports the parallelization.

The parallelization of the GLEAMS decision tree procedure differs from any of the above-mentioned approaches in the fact that GLEAMS assumes the training set size can fit the RAM of a single computer. Indeed, the algorithm starts by synthetically generating an arbitrarily large set of examples, i.e. the Sobol points, and computing their prediction using the input ML model. Then, during the computation of any candidate leaf, if necessary, other examples can be generated in order to improve the accuracy of the explanation. Hence, it is not crucial for GLEAMS parallelization to resort to horizontal partitioning: the communication overhead which characterizes the solutions based on this domain decomposition can be easily avoided.

Another key difference consists in the $\hat{Y}$ value assigned to each leaf: general decision tree algorithms assign the average $\bar{y}$ value inside the node, while GLEAMS stores the hyperplane formula, in the form of feature coefficients. Through the formula, GLEAMS assigns different $\hat{y}$ values at different points in the node. As described in Section 2, in order to compute the hyperplane, the algorithm needs all the values of the features and the target. This difference automatically hinders a domain decomposition that assigns just one feature and the target to each computing processor as in pure vertical data parallelism.

4. Parallelizing GLEAMS

Since we resort to Ray in order to parallelize GLEAMS, in the following we start by describing the key characteristics of this distributed computing engine and then move on to depict how we adapted GLEAMS to Ray's programming paradigm.

4.1. Ray programming paradigm

Ray[3] is a unified framework for scaling AI Python applications. It offers a toolkit of libraries for ML computation designed for the end-used, as well as a simple, yet powerful, Core API that can be used to parallelize custom applications on a local machine or a cluster. W.r.t other, more popular frameworks such as Apache Spark,[4] Ray shows several key advantages that made it preferable for the parallelization of GLEAMS: (i) as we describe in the following, Ray offers a simpler architecture and programming paradigm than Spark, making it more suitable to implement unified solutions meant to be launched on a multiprocessor machine, as well as on a large cluster; (ii) Ray offers tools for the automatic provisioning, launch and auto-scale on GCP, Amazon EC2, MS Azure and Kubernetes,[5] thanks to which, the parallelized application can be quickly deployed on a cluster of several machines with virtually zero effort; (iii) Ray is also able to maintain a smaller state for each running application and to provide speedy scheduling of light tasks—crucial points of advantage over Spark for GLEAMS' parallelization as well as for various ML applications; (iv) while Spark envisages shared immutable objects managed only by the master node – a choice that can sometimes create bottlenecks – in Ray this task can be carried out by any node; (v) instead of being written in Scala as Spark, Ray Core is Python code supported by an underlying C++ layer that handles the heaviest tasks, making it extremely performant. Indeed, Ray offers higher performance than other distributed processing engines according to various comparisons [7,66,67]. For all these reasons, our choice for the parallelization of GLEAMS fell on Ray distributed engine.

[3] https://www.ray.io/

[4] https://spark.apache.org/

[5] https://docs.ray.io/en/latest/ray-overview/ray-libraries.html#the-ray-ecosystem

Table 2
Classification of related works on parallel decision tree-building algorithms.

Approach	Domain decomposition	Tree construction strategy	Implementation Framework	Decision tree type	Distributed platform
PDT [31]	HD	synch.	custom, 3-phase	ID3, C4, CART	not implemented
Pearson [30]	VD	part.	master/worker	general	custom
Bowyer et al. [37]	HD	part.	ensemble (majority voting)	C4.5	custom for ASCI Red parallel
DTD [28]	T	part.	master/worker	C4.5	custom
DP-rec [28]	HD	synch.	custom	C4.5	custom
DP-att [28]	VD	synch.	custom	C4.5	custom
ScalParC [34]	HD	part.	pre-sort	general	custom MPI
SLIQ [33]	HD	part.	pre-sort	general	custom
SPRINT [33]	HD	part.	pre-sort	general	custom
Amado et al. [27]	HD	synch.	custom	C4.5	custom MPI
Srivastava et al. [48]	T+HD	hybrid	master/worker	general	custom MPI
SPIES [35]	HD	synch.	sampling & histogram building	general approx. tree	FREE- RIDE
PLANET [53]	HD	part.	ensamble + MR	general	Google MapReduce
SPDT [36]	HD	synch.	sampling & histogram building	general approx. tree	custom MPI
Dai et al. [52]	HD	synch.	MR	C4.5	Hadoop
PV-Tree [40]	HD	part.	ensamble (2-stage voting)	general	custom
Mu et al. [50]	HD	synch.	MR	C4.5	Hadoop+ WEKA
Mu et al. [51]	HD	synch.	MR	PCC-tree	WEKA
Segatori et al. [55]	T+HD	part.	MR	Fuzzy DTs	Spark
Mu et al. [56]	HD	synch.	MR	Fuzzy rule-based DTs	Hadoop
SySM [42]	HD	part.	ensamble	CART, J48	(*)
Fan et al. [41]	HD	part.	ensamble	general	(*)
Nasridinov et al. [57]	T+VD	part.	single-GPU	ID3	CUDA
Strnad et al. [58]	T+VD	part.	single-GPU	CART	CUDA
Jurczuk et al. [59]	HD	synch.	single-GPU	EA/classification	CUDA
Jurczuk et al. [60]	HD	synch.	single-GPU	EA/regression	CUDA
Jurczuk et al. [61]	HD	synch.	single-GPU	EA/classification	CUDA
Jurczuk et al. [64]	HD	synch.	multi-GPU	EA/classification	CUDA
Jurczuk et al. [62]	HD	synch.	single-GPU	EA+ML/regression	CUDA
Jurczuk et al. [63]	HD	synch.	single-GPU	EA/classification	CUDA
Costa et al. [65]	HD	synch.	single-GPU ensamble	EA/classification	CUDA

(*) Does not specifically address parallelization but proposes an ensamble strategy that can be easily applied to HD.

Like Apache Spark, Ray's architecture is based on a master–slave paradigm. As shown in Fig. 2, both the master, called Head node, and the slave, called Worker, run a Raylet process for daemon coordination and object management, and an Object Store, which works as a distributed shared-memory space for ray applications, allowing communications between processes on the same machine. Finally, the Head node also runs the Global cluster store: a Redis in-memory database that contains information about the cluster.

The Python process that starts to execute the script containing Ray code is called *driver* process. This process then either connects to a Ray cluster or generates a new local one to handle the remote function calls by spawning *worker processes*. For both the driver and the workers, Ray employs processes rather than threads, to avoid incurring Python's GIL mutual exclusion. By default, each node hosts a number of workers equal to the number of logical CPU cores of the node. These workers form a pool used to schedule tasks. As shown in Fig. 2, both the driver and the workers can be allocated either on the Head node (if enough cores are available), or another Worker machine of the cluster.

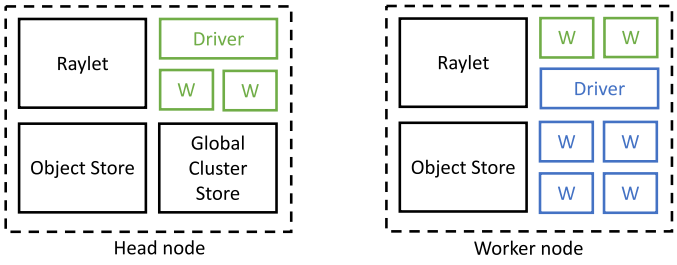

Fig. 2. Architecture of Ray Core.

Ray's programming paradigm is based on three key concepts: *tasks, actors* and *remote objects*. Ray *tasks* allow arbitrary functions to be executed asynchronously on separate Python worker processes. To define a task a user simply needs to annotate a function with the `@ray.remote`

decorator, and the task is executed by adding `.remote()` to the function call like this:

```
# asynchronous remote call
future = func.remote(params)
# blocking call to get the result
result = ray.get(future)
```

The remote call schedules the execution of the function on a Ray worker, either on the local machine or on a remote one in the case of a cluster. Even in the case of a cluster, the worker could still be in the local machine unless all the available CPUs on that machine are already busy. In order to obtain parallelism, the user simply invokes multiple asynchronous `.remote()` functions before waiting with `.get()` for the list of remote results references.

Ray *actors* are essentially stateful workers, created by annotating classes with the `@ray.remote` decorator and instantiated with the `.remote()` form. Then, all its methods can be called with the `.remote()-.get()` paradigm. Each actor instance is assigned a single dedicated worker on which are scheduled all the method calls of that instance. The method calls are guaranteed to be executed sequentially and in order, which makes the management of the internal state simple to handle. It is possible to create special asynchronous actors that forgo the order and sequential guarantee, in which case it is the developer's responsibility to maintain the coherence of the internal state.

Tasks and actors produce and process *remote objects* that can be saved in the Object Store of any node of the Ray cluster. Actually, a remote object can reside on one or more nodes' Object Store, regardless of who has the object reference, but in general, the data is only transferred to the nodes that actually need it, avoiding unnecessary network overhead.

4.2. GLEAMS on ray

Employing a profiler, it is possible to see how the most time-consuming part of GLEAMS' code is related to the computation of the best-split point and its score for each dimension. Therefore, our initial goal was to parallelize this task following a vertical data parallel approach.

4.2.1. Parallelization of the best-split point computation

GLEAMS input dataset is composed of a set of examples, each one representable through a d-dimensional point. Each dimension is a different variable (aka feature) of the domain. The algorithm needs to figure out along which dimension it is best to split the subdomain so that the points on each side present the most similar features. To do so, the best-split points along all the dimensions are computed together with a score. The score is then used to select the best dimension and, with it, the best-split point overall. In each node of the tree, the computation of the split point and score for each dimension is independent of the others and can be performed in parallel, thus following a vertical data strategy. We will refer to this solution as VD-GLEAMS in the following. As shown in Fig. 3 VD-GLEAMS envisages that the computation of the tree structure (black elements in Fig. 3) is carried out by the Ray driver process, while the computation of the split point is computed on a dedicated worker for each dimension (coloured elements in Fig. 3).

Thanks to Ray's programming paradigm, this parallelization did not require many changes apart from annotating the function GetBestSplitForFeature (responsible for the split-point computation for each dimension) with the `@ray.remote` decorator and changing the loop where it was invoked with the `.remote()-.get()` paradigm. As shown in Algorithm 4, this means, firstly, asynchronously calling the remote function GetBestSplitForFeature for each dimension (line 7) to get a reference to the best-split point and score on that dimension; then triggering the resolution of the references (line 9) to obtain the parallelly computed results.

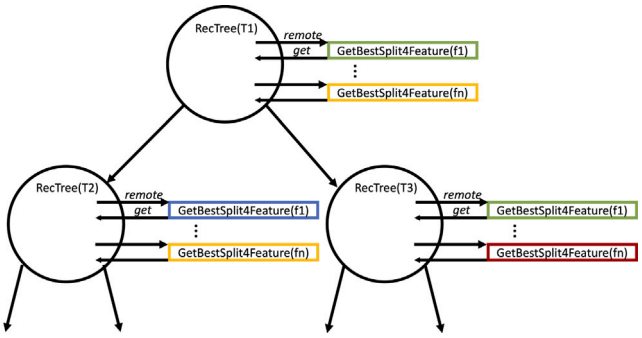

Fig. 3. Architecture of VD-GLEAMS. Different colours correspond to executions on different Ray processes.

As anticipated, differently from traditional decision tree-building algorithms, the computation of the split points in GLEAMS is based on the computation of a cumulative score process. Therefore, to compute the best split along a certain dimension, the values of the points for that dimension are not sufficient. The algorithm requires the values of the points for *all* the model features. From the point of view of parallelization, this entails the need for each worker process to access very large arrays containing all feature and target values for the initially generated input points. To efficiently manage these large data the calls to `ray.put()` in lines 4 and 5 preemptively place the feature and target values on the Ray Shared Object storage and return a reference to them. The references are then passed to the remote functions in order to avoid duplicating the arrays for each domain dimension.

The level of parallelism of this vertical data parallel solution is nonetheless limited by the number of features of the examined model.

Algorithm 4 Parallel computation of the best split for a given tree's node: VD-GLEAMS.

Input: X: set of Sobol points in the current node; Y: values of f on X; d: number of features in the input.
Output: j^*: feature to split; t_{j^*} value of the split

1: **procedure** ParallelGetBestSplit($X, Y, \hat{f}$)
2: $\hat{Y} \leftarrow \hat{f}(X)$
3: $S \leftarrow$ GetScores$\hat{f}, \hat{Y}, X$
4: $X_{\texttt{ref}}$=`ray.put()`
5: $Y_{\texttt{ref}}$=`ray.put()`
6: **for** $j = 1$ to d **do**
7: $(m_j, t_j)_{\texttt{refs}}$=GetBestSplit4Feature`.remote(`
8: $X_{\texttt{ref}}, Y_{\texttt{ref}}, S, j$)
9: $[(m_j, t_j)]$=`ray.get(`$(m_j, t_j)_{\texttt{refs}}$`)`
10: $j^* \leftarrow \text{argmax}_{j=1,\dots,d}\, m_j$
11: t_{j^*}, value of the split
12: **return** (j^*, t_{j^*})
13: `@ray.remote`
14: **procedure** GetBestSplit4Feature(X, Y, S, j)
15: sort $[X, Y, S]$ according to feature j
16: $B^{(j)} \leftarrow$ CumulativeScoreProcess$S$
17: $m_j \leftarrow \max_{i=1,\dots,N} \left\| B^{(j)}(i) \right\|_1$
18: $t_j \leftarrow \text{argmax}_{i=1,\dots,N} \left\| B^{(j)}(i) \right\|_1$
19: **return** m_j, t_j

4.2.2. Parallelization of the tree's structure

An alternative solution is the task parallel approach, where the domain decomposition follows the structure of the decision tree by assigning different tree nodes (or sub-trees) to different parallel processes as shown in Fig. 4. We refer to this solution as T-GLEAMS. Using the Ray paradigm, we annotate the function RecTree (responsible

Algorithm 5 Task parallel computation of the global surrogate model used by GLEAMS: T-GLEAMS.

Input: f: black-box model ; $\mathcal{X}$: boundaries of the input space
Output: T: decision tree explanation for f

```
1: procedure GlobalSurrogate(f, 𝒳)
2:     generate X = {x_1, ..., x_N} Sobol points in 𝒳
3:     y_i ← f(x_i) ∀i ∈ {1, ..., N}
4:     Y ← {y_1, ..., y_N}
5:     T_ref ← RecTree.remote(X, Y)
6:     loop=asyncio.get_event_loop()
7:     T ←loop.run_until_complete(
8:                         resolve_tree(await T_ref))
9:     return T
10: @ray.remote
11: procedure RecTree(X, Y)
12:     if |X| < 2^n_min then
13:         𝒳 ← boundaries of the input space of X
14:         return GlobalSurrogate f, 𝒳
15:     f̂ ← LinReg X, Y
16:     if R²(X, f̂) > ρ  then return T as leaf
17:     (j, t_j) ← GetBestSplit X, Y, f̂
18:     T ← Node X, Y, j, t_j
19:     (X^ℓ, Y^ℓ, X^r, Y^r) ← SplitData X, Y, j, t_j
20:     T.ℓ_ref ← RecTree.remote(X^ℓ, Y^ℓ)
21:     T.r_ref ← RecTree.remote(X^r, Y^r)
22:     return T
23: async def resolve_tree(T)
24:     if T.terminal then
25:         return T
26:     else
27:         T.ℓ,T.r = await asyncio.gather(T.ℓ_ref, T.r_ref)
28:         ▷ Remove references to remote objects to allow their
    cleanup:
29:         T.ℓ_ref = None
30:         T.r_ref = None
31:     T.ℓ,T.r = await asyncio.gather(
32:                 resolve_tree(T.ℓ), resolve_tree(T.r))
33:     return T
```

for recursively generating the tree from the root to leaves) with the `@ray.remote` decorator as shown in line 10 of Algorithm 5, and we turn its invocation into an asynchronous `ray.remote()` call (line 5). Also, inside the RecTree procedure all the recursive calls need to be turned into asynchronous ones (lines 20 and 21).

In this way, for each node of the tree, a remote task is invoked. Given the recursive nature of the RecTree each node might then have to invoke further child remote functions. The use of `ray.get()` needs to be avoided in this case, because it causes each node to wait for the completion of its child's asynchronous call. Indeed, as mentioned in the previous subsection, each remote task, once scheduled, is assigned to a Ray worker process. The total number of workers is supposed to be limited and in the same number as the number of logical cores present in the Ray cluster. Nonetheless, a task that is blocked waiting on the `ray.get()` call is also keeping the worker blocked. In these situations, Ray avoids deadlocks by instantiating new workers, so that they may handle the still-pending tasks. This is a good choice in general, but in our case, as the depth of the tree might be significant, it may overwhelm the cluster with the spawning of several workers, each one just waiting for the node's children to complete.

To avoid this issue, in our solution, each task submits the asynchronous children's remote tasks but does not wait on them with `ray.get()`. Instead, the task returns immediately with a node containing unresolved remote references. Then, to resolve the generated tree of

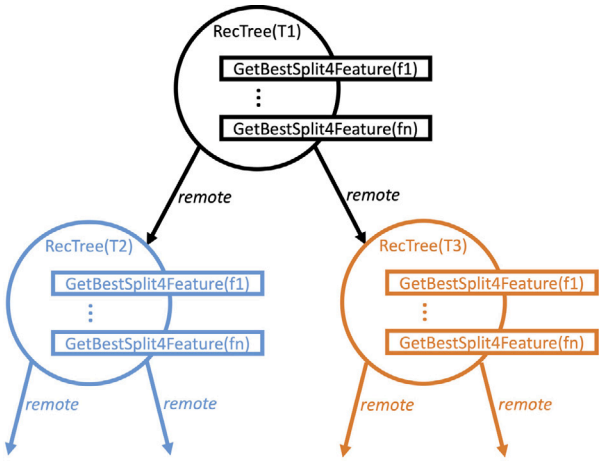

Fig. 4. Architecture of T-GLEAMS.

references a recursive asynchronous function is implemented through the `asyncio` Python standard library[6] (line 7 of Algorithm 5) which is integrated into Ray.

While the underlying operating system keeps seeing only one process, `asyncio` spawns numerous concurrent library-level "pseudo-threads", called *coroutines*. Coroutines are supervised by a master scheduler, called *event loop*. It schedules a coroutine from a pool of tasks that needs to be executed, and then, whenever the coroutine blocks waiting for a certain operation to be executed (in our case, waiting for the resolution of a node reference), it is suspended and the loop is freed to start a new coroutine from the pool. In this way, even if one of the tree's branches is still stuck in computation, the asynchronous function can still resolve other parts of the tree while it waits. In other words, each RecTree task generates two child tasks (except in the case of a leaf node) and concludes avoiding the overcrowding of workers; then an asynchronous recursive function (`resolve_tree()` called in line 7) is used to resolve the tree of remote references efficiently and concurrently.

Like any task parallel approach to decision tree building, the maximum amount of parallelism obtainable with this solution depends on the structure of the tree.

4.2.3. Combining the two parallelization strategies

In order to increase the maximum level of parallelism, the two solutions illustrated above can be merged together. The idea is to generate a new worker process to handle the computation of each child tree node, each of which would then spawn further worker processes to compute the best-split point in parallel. In practice, this can be easily realized by changing line 17 of Algorithm 5 so as to call the ParallelGetBestSplit of Algorithm 4 instead of the original the GetBestSplit. We call this solution Mixed Vertical Data and Task parallel GLEAMS (VDT-GLEAMS). Its structure is exemplified in Fig. 5 where different colours are used for different algorithm parts to highlight that they are executed on different Ray processes.

In principle, this approach could obtain a higher level of parallelism than the previously presented ones because it combines the benefits of both task and data domain decompositions. Nonetheless, the nested nature of the resulting parallelism is problematic. Indeed, any worker process executing a RecTree operation needs the result of the parallel split-point computation to compare them and select the best one (and then divide the remaining domain into the two child domains). This means that each worker process executing RecTree will block during the parallel computation of the split-points (on line 7 of Algorithm 4).

[6] https://docs.python.org/3/library/asyncio.html

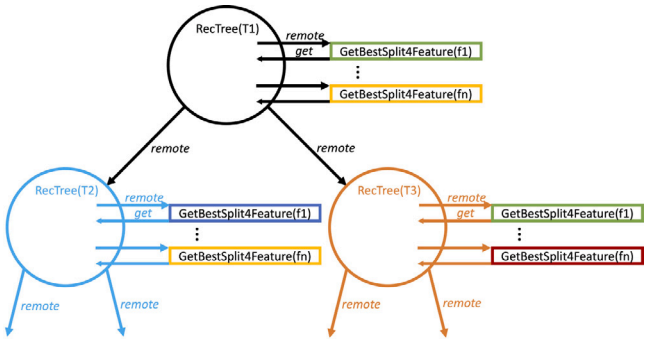

Fig. 5. Architecture of VDT-GLEAMS.

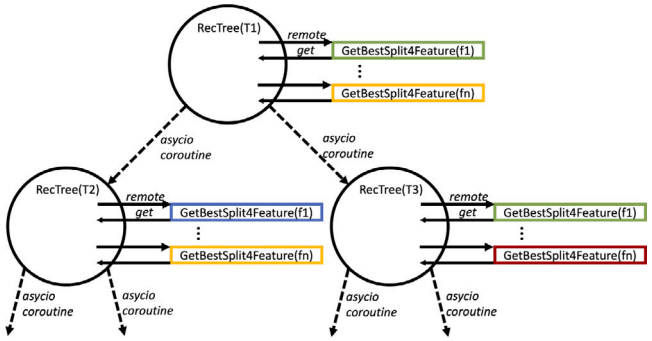

Fig. 6. Architecture of VDTa-GLEAMS.

The problem could be observed better using the Ray timeline command.[7] Fig. 7(a) shows the simple timeline of a 4-core Ray cluster executing on an 8-core machine with hyperthreading. It is easy to see how the system spawns far more than four processes to run, each of which shows long inactivity periods. This behaviour is precisely due to the RecTree processes waiting for the parallel computation of the split points conducted by other worker processes.

An alternative solution to combine the benefits of both the task-parallel and vertical data-parallel approaches is illustrated in Algorithm 6. Here, task parallelism is not realized through multiple calls to separate Ray workers, but rather through an `asyncio` event loop implemented on Ray's driver process. As a result, the distributed engine will spawn multiple workers for the parallel computation of the best-split point whereas the tree's structure computation is realized with `asyncio` coroutines, which concurrently take care of the tree's different nodes on the driver process. We will refer to this solution as VDTa-GLEAMS. Black elements in Fig. 6 represent the algorithm parts executing on the driver, while the research of the best-split point is carried out on different Ray workers.

Traditionally `asyncio` computation is more suited to handle IO-bound operations, while the computation of the tree's structure is certainly ascribable to a CPU-bound task. Nonetheless, the idea of Algorithm 6 originates from the observation that, comparatively, the execution time of the Ray remote tasks computing the best-split points was much longer than the execution time of all the other operations. Therefore, we speculated that the computation of the tree's structure could be conducted by a single process, concurrently handling multiple calls to remote best-split points computations, and efficiently waiting on the results thanks to the `asyncio` mechanism. Fig. 7(b) shows the timeline of this alternative solution executing on a 4-core Ray cluster

[7] Ray timeline generates a JSON file containing the execution information of the job, that can be examined, for example, by using the chrome://tracing utility.

Algorithm 6 Computation of the global surrogate model used by GLEAMS with `asyncio`: VDTa-GLEAMS.

Input: f: black-box model ; $\mathcal{X}$: boundaries of the input space **Output:** T: decision tree explanation for f

procedure GlobalSurrogate($f, \mathcal{X}$)
  generate $X = \{x_1, \ldots, x_N\}$ Sobol points in $\mathcal{X}$
  $y_i \leftarrow f(x_i)\ \forall i \in \{1, \ldots, N\}$
  $Y \leftarrow \{y_1, \ldots, y_N\}$
  `loop=asyncio.get_event_loop()`
  $T \leftarrow$ `loop.run_until_complete(`RecTreeX, Y`)`
  return T
procedure `async` RecTree(X, Y)
  if $|X| < 2^{n_{min}}$ **then**
    $\mathcal{X} \leftarrow$ boundaries of the input space of X
    return GlobalSurrogate$f, \mathcal{X}$
  $\hat{f} \leftarrow$ LinRegX, Y
  if $R^2(X, \hat{f}) > \rho$ **then return** T as leaf
  $(j, t_j) \leftarrow$ ParallelGetBestSplit$X, Y, \hat{f}$
  $T \leftarrow$ NodeX, Y, j, t_j
  $(X^\ell, Y^\ell, X^r, Y^r) \leftarrow$ SplitDataX, Y, j, t_j
  $T.\ell, T.r \leftarrow$`await asyncio.gather(`
    RecTreeX^ℓ, Y^ℓ, RecTreeX^r, Y^r`)`
  return T

deployed on an 8-core machine with hyperthreading. Compared to Fig. 7(a) it is easy to see that the number of launched workers remains limited to 4. Furthermore, each line appears as completely filled, thus providing a preliminary confirmation to the initial assumption of a more efficient resource usage of VDTa-GLEAMS w.r.t. to VDT-GLEAMS.

To confirm these intuitions and to further investigate the performance of the proposed approaches, we conduct a set of experiments on all the described parallel variants, as reported in the following section.

5. Performance evaluation

The objective of the analysis we are going to discuss in this section is to gain a quantitative understanding of the performance improvements that can be obtained by parallelizing GLEAMS according to different strategies. We remark that we are not interested here in evaluating the accuracy of the explainability method, which has been already studied in [6]. Indeed, since none of the parallel solutions proposed alter GLEAMS's computing steps, the measure of the algorithm's ability to provide an explanation to a ML model would be independent of the solution being parallel or otherwise.

5.1. Experimental setup

We test the approaches on a single-computer setup and a cluster of several interconnected machines. The considered single computer is equipped with an AMD Ryzen 7 3700x 8-core physical machine with hyperthreading (i.e., up to 16 logical cores) working at 3.6 GHz with 32 GB RAM.

The cluster is composed of 98 identical machines. Each one is equipped with a 3.8 GHz Intel(R) Core(TM) i3-10305 CPU with 4 cores, hyperthreading (i.e., up to 8 logical cores) and 16 GB RAM. The machines are interconnected by a 1 Gbps Ethernet network.

5.2. Evaluation approach

GLEAMS starts by taking a ML model as input and synthetically generating the Sobol points useful to provide an explanation.

We employ the regression benchmark problem Friedman#1, described in [69,70], as the ML model over which GLEAMS needs to fit the

Fig. 7. Ray timeline of VDT-GLEAMS in Fig. 7(a) and VDTa-GLEAMS in Fig. 7(b) executing the same workload on a 4-cores-limited machine [68].

binary tree. In order to test the scalability of our parallel approaches, we evaluate their performance on synthetic input datasets of increasing size by varying two variables: (i) the dimension d of the domain (i.e., the number of features of each of the points in the input dataset); and (ii) the number of generated Sobol points $|X| = 2^m$ (by increasing the value of the exponent m). Obviously, different sizes of the input array (m) and different numbers of features (d) have a great impact on the execution profile of the algorithm, thus multiple configurations were tested. d is a factor that depends on the problem that the user is trying to analyse. Most problems that are of interest for GLEAMS algorithm will not have a very high number of features, most likely between 5 and 30. Nonetheless, higher values of d are included in these tests to observe the behaviour of the system in extreme conditions. The parameter m, on the other hand, can be changed by the user: high values of m offer increasing granularity and accuracy of GLEAMS' results, at the cost of a higher resource utilization and execution time. Realistic usage values for m are between 14 and 20 [10].

For each dataset, the generation process operates by sequentially computing 2^m points. We choose 9 different integer values for m, i.e., $12 \leq m \leq 20$. Each of these points has d dimensions, and we vary d in $\{5, 10, 20, 50, 100\}$. For each dimension, the generating procedure chooses a floating-point value between 0 and 1 following the prescriptions of Sobol [23]. Therefore, we generated $9 \times 5 = 45$ different datasets for testing.

To make the benchmark run computationally more complex, a random Gaussian noise of fixed seed and standard deviation of 1 was included in the generation of the Sobol points. Indeed, in our setup, Friedman#1 would be the black box model that GLEAMS must explain. The added noise makes Friedman#1 more complex, challenging the GLEAMS procedure on an arguably computationally intensive problem.

The resulting dataset is given as input to the original sequential implementation of GLEAMS and the four parallel approaches presented in this work (i.e., VD-GLEAMS, T-GLEAMS, VDT-GLEAMS and VDTa-GLEAMS).

The code of all GLEAMS' versions as well as the code to reproduce these tests is hosted on GitHub.[8]

The tests are grouped into the following three sets.

- *Group 1* - We investigate the speedup obtained thanks to parallelization on a single multi-core machine and a cluster of several nodes. Furthermore, we deepen the scalability feature of the various solutions for increasing input sizes, by varying both d and m of the examples dataset. In this regard, two possibilities exist: *strong scalability* highlights how the execution time varies with the number of available processors, while *weak scalability* evaluates the computation time when the load on each processor is kept constant and we increase both the number of processors and the problem size [71]. The nature of GLEAMS' workload does not allow a consistent way to increase the size of the problem together with the number of processors, thus preventing a reliable weak scalability test. Therefore, our evaluation focuses on strong scalability only.
- *Group 2* - Albeit none of the proposed implementations improve GLEAMS' ability to explain ML models, it is interesting to study

[8] https://github.com/giorgiovisani/Glob_Lime

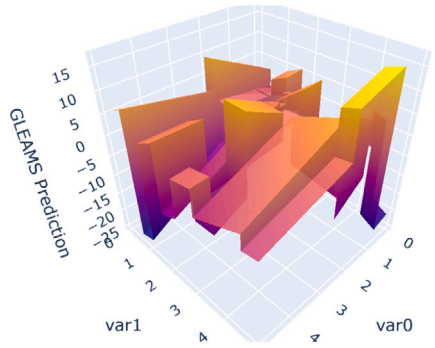

Fig. 8. Example (taken from [6]) of a synthetic surface model (with just two input features to allow visualization) like the one provided in input to Group 2 tests.

how the reduction of the execution time due to parallelization relates to the accuracy of GLEAMS in reconstructing a ML model. To this end, instead of a black-box model, we provide a known, synthetic ML model as input f, and we test the accuracy (of reconstructing such a model) reached in a certain time by the sequential and the parallel implementations. The synthetic input model is a discontinuous piecewise-linear model with 10 input features and arbitrary coefficients, which is a generalization of the surface produced by a Random Forest regressor. Fig. 8 (taken from [6]) shows a simplified example of the synthetic input model we consider. To estimate the accuracy of the reconstruction, we employ R^2 as a measure of how close the surrogate model $\hat{f}$ is to the true input model f. Since f is composed of several hypervolumes with different sizes, the R^2 distance between f and $\hat{f}$ is actually computed as an average of the R^2 values in each hypervolume, weighted based on the hypervolumes' size.

- *Group 3* - We briefly investigate the gains that can be obtained in terms of reducing energy consumption when a parallel approach is employed for GLEAMS. Indeed, the adoption of a parallel approach implies the utilization of multiple cores during the execution, which may translate into an increase in the consumed power unless the technique is able to counterbalance this by significantly reducing the computation time. As energy saving is becoming a major concern in many fields, it becomes crucial to investigate if the proposed parallel technique is actually able to reduce the time to compute the solution so much that it can also bring benefits from the point of view of power consumption. To this end, we enrich the implementations with CodeCarbon[9] directives, which allow us to provide an estimation of the total energy consumed during the computation for increasing input datasets.
- *Group 4* - We compare the execution times of parallel GLEAMS against other explainability approaches, namely SHAP [3] and LIME [4]. These methods are the closer competitors to GLEAMS and are considered the state-of-the-art explainability techniques in the context of tabular data. However, LIME is a local technique: to obtain a global explanation, we need to apply it to each point of the dataset. SHAP behaves similarly, with the additional benefit that local explanations can be aggregated in order to obtain a global one. Nonetheless, both techniques have proven to be rather fast. It is therefore important to compare their computation time with GLEAMS, which is able to achieve a local and global explanation at the same time.

[9] https://codecarbon.io

5.3. Results

Concerning the tests of *Group 1*, Fig. 9 focuses on the execution on a single machine and shows the speedup of each parallel solution (VD-GLEAMS, T-GLEAMS, VDT-GLEAMS and VDTa-GLEAMS) over the sequential one (GLEAMS), for different input dataset sizes generated by different combinations of m and d.

The solution offering the best improvement overall appears to be the asynchronous one VDTa-GLEAMS. Some of the other solutions, in particular VD-GLEAMS and VDT-GLEAMS, are also competitive in some cases. As expected, VD-GLEAMS (which parallelizes only the split-point computation) offers lower performance than VDTa-GLEAMS because it cannot start the computation of multiple tree nodes at once. The combined parallel split-point and recursive tree solution VDT-GLEAMS is perhaps the best runner-up, as it is also able to compute multiple tree nodes at the same time like the async solution VDTa-GLEAMS, but it shows some losses in the case of small m values, i.e., when the input dataset is smaller (remember that the number of Sobol points is given by 2^m). This behaviour is most probably caused by the worker's initialization overhead. VDT-GLEAMS uses `ray.get()` blocking operations, which require the creation of new workers to avoid deadlocking. As the number of Sobol points increases, the computational load assigned to each worker increases and the whole execution is less influenced by the initialization overhead. It is also relevant to point out that all the solutions except for T-GLEAMS show better speedups with a higher number of domain dimensions d. This is consistent with the fact that, for VD-GLEAMS, VDT-GLEAMS and VDTa-GLEAMS, d directly defines the amount of parallelism available to the solution, whereas the degree of parallelization of T-GLEAMS is only influenced by the tree's structure. Therefore, for VD-GLEAMS, VDT-GLEAMS and VDTa-GLEAMS, higher values of d imply a higher number of parallel tasks, hence better utilization of the multi-core system.

Fig. 10 shows the speedup of the parallel solutions w.r.t. the sequential version of GLEAMS when executing on a cluster of 98 interconnected machines. For most of the conducted tests, the total number of computation units (784 cores) is much larger than the maximum achievable parallelism, thus allowing us to observe the behaviour of the solutions in case of unbounded computation resources available. Analogously to what happens in the single-node setup, the speedup of T-GLEAMS remains almost constant for all input sizes while all three other solutions increase in speedup as d increases.

As expected, VD-GLEAMS usually shows a limited speedup w.r.t. VDT-GLEAMS and VDTa-GLEAMS but there are some outliers in which, on the contrary, VD-GLEAMS performs better than VDT-GLEAMS. This contradicts the logic by which, being based on the parallelization of both the tree and the split-point computation, VDT-GLEAMS should always perform better than VD-GLEAMS (which parallelizes only the split-point). Nonetheless, as we have already observed in the case of a single-machine execution, VDT-GLEAMS causes Ray's spawning of multiple workers to resolve the potential deadlocks caused by the blocking `ray.get()` calls. The overhead of initialization of all spawned workers is therefore most probably the cause of these performance degradations.

On the other hand, we can also observe that, for low values of d, there are some cases in which the speedup of the async-based solution VDTa-GLEAMS performs slightly worse than VDT-GLEAMS. This could be attributed to VDT-GLEAMS having the advantage of an actually parallel execution of the functions handling each tree node. The advantage is then lost when d increases as this implies a higher-than-necessary number of workers to be spawned and machines to be involved for VDT-GLEAMS. However, from the unbounded-resources tests overall, we could conclude that the async-based solution VDTa-GLEAMS is the one with best scalability performance, probably due to higher resource efficiency.

In order to confirm this conclusion, we also perform a strong scalability evaluation on the cluster. The results are illustrated in Fig. 11 for three different input sizes $(m, d) = \{(16, 20), (18, 20), (18, 50)\}$. The choice

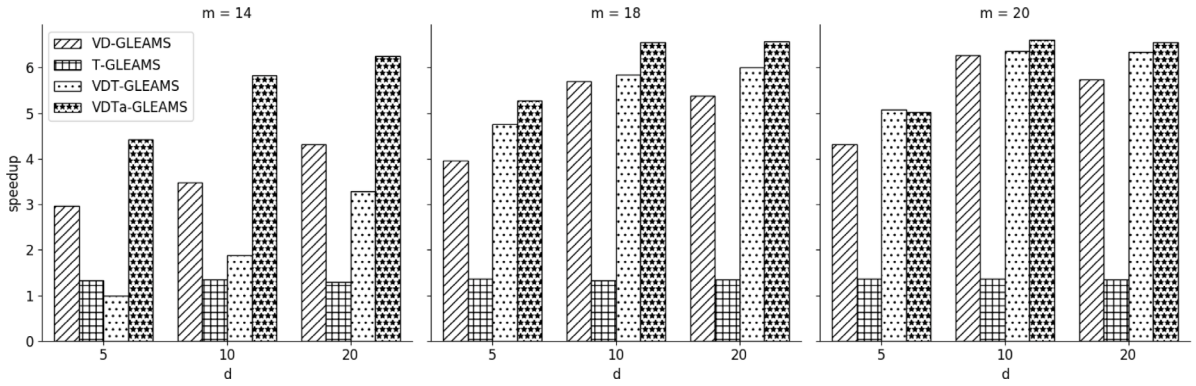

Fig. 9. Speedup of the parallel solutions w.r.t. the sequential implementation of GLEAMS on a single multi-core machine.

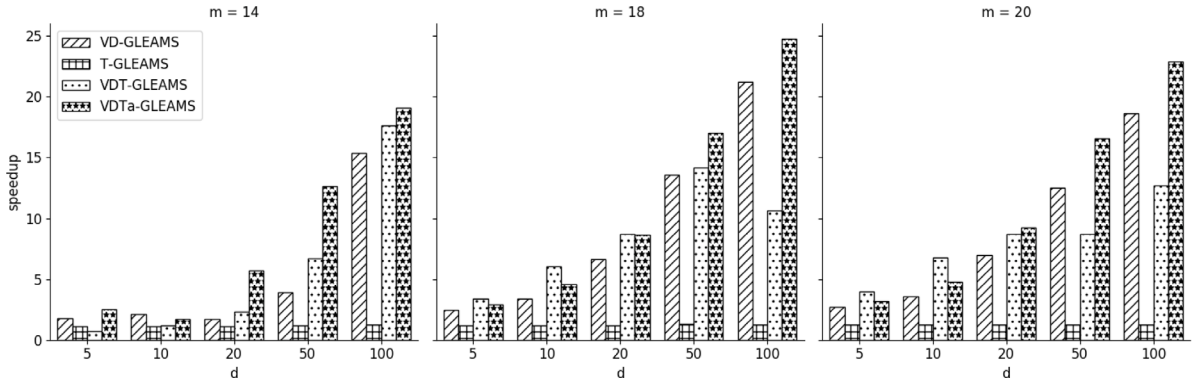

Fig. 10. Speedup of the parallel solutions w.r.t. the sequential implementation of GLEAMS on a cluster of interconnected machines.

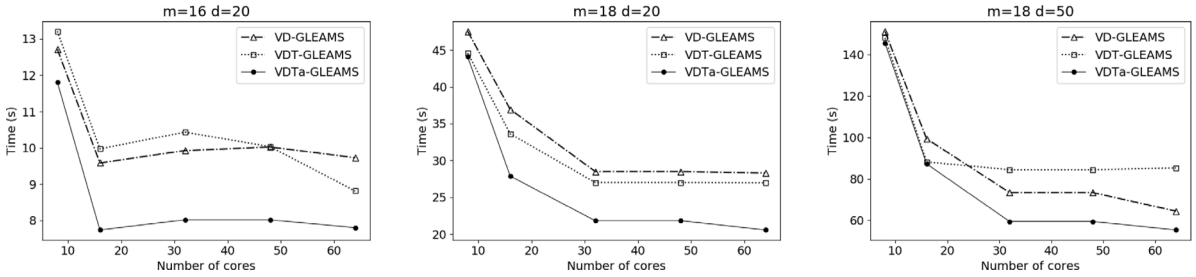

Fig. 11. Strong scalability evaluation of all four proposed parallel solutions.

of values for m is motivated by the will to evaluate GLEAMS in the best conditions under the point of view of the explanations' accuracy. On the other hand, $d = 20$ is the average number of features that the algorithm is expected to handle in real-life usecases. The additional $(m, d) = (18, 50)$ combination is included to stress the system's capabilities.

As prescribed by strong scalability, for each chosen (m, d) combination we increase the number of computing units involved in the computation. In all tests, Ray daemon is limited to using only 4 CPUs per machine. For the sake of the graphs' readability, we exclude T-GLEAMS from these tests because, as expected, its execution time is by far the most relevant one for all three input dimensions. For all the proposed parallel solutions, it is also possible to appreciate a graceful decrease in the computation times as more CPUs are available, thus confirming their good scalability performance. However, the graphs remark how VDTa-GLEAMS is always the best solution in terms of computation time.

For this reason, in *Group 2* tests, we focus on VDTa-GLEAMS and we compare the quality of the solution provided in a certain time with the one obtained by Seq-GLEAMS in the same amount of time. To this end, we run 6 tests for both implementations with an increasing number (2^m) of input Sobol points. Intuitively, the higher is the value of m, the more accurate is GLEAMS' solution, and the higher is also the execution time.

Table 3 reports the considered values of m, and the corresponding number of leaves and depth of the tree computed by (any implementation of) GLEAMS. Fig. 12 shows instead the trends of the R^2 metric provided by VDTa-GLEAMS and Seq-GLEAMS for increasing execution time. The graph highlights how, if we consider the same amount of time, the adoption of the parallel solution VDTa-GLEAMS comes with sensibly higher accuracy. For example, in 20 s Seq-GLEAMS reaches $R^2 = 0.4$, whereas VDTa-GLEAMS provides $R^2 = 0.6$.

In *Group 3* tests, we focus again on VDTa-GLEAMS execution and we investigate the gains it can bring from an energy consumption standpoint—a matter of growing importance nowadays.

Fig. 13 shows the comparison of the energy consumed by the original sequential implementation of GLEAMS (Seq-GLEAMS) and the parallel asynchronous version VDTa-GLEAMS for increasing values of m. The trends clearly show that, as we augment the number of generated Sobol points, the increase of power utilization of the parallel solution remains contained w.r.t. that of Seq-GLEAMS. This highlights a

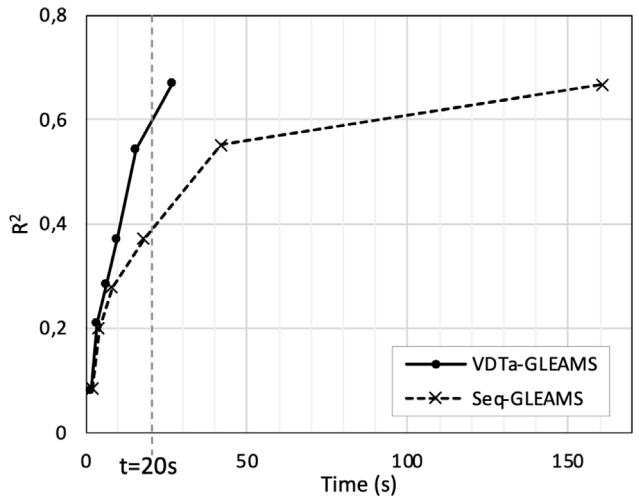

Fig. 12. Accuracy (measured in terms of R^2) w.r.t. execution time of Seq-GLEAMS and VDTa-GLEAMS.

Table 3
Values of m considered for the tests of Group 2 and the corresponding number of leaves and levels of the generated tree.

m	Leaves	Depth
12	146	16
13	269	18
14	478	18
15	853	20
16	1229	23
17	1753	24

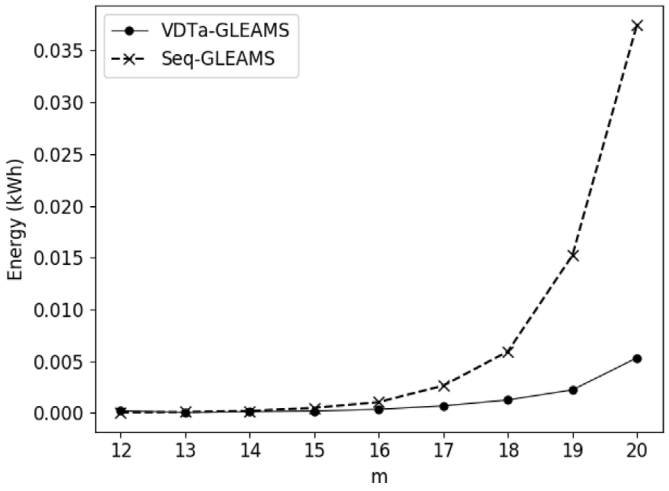

Fig. 13. Energy consumed by the standard sequential implementation of GLEAMS compared to the parallel VDTa-GLEAMS.

desirable feature of VDTa-GLEAMS: despite the employment of multiple cores, which in principle could increase the power utilization, the reduction of the execution time is substantial enough to translate into a significant decrease of the energy consumption.

Regarding *Group 4*, we choose VDTa-GLEAMS as our reference parallel implementation of GLEAMS and we compare it with Seq-GLEAMS and other existing explainability methods in terms of computation time when we increase the number of points to be explained. We remark that, in this group of tests, we do not evaluate the accuracy of the methods involved, as it has already been studied in [6]. The results

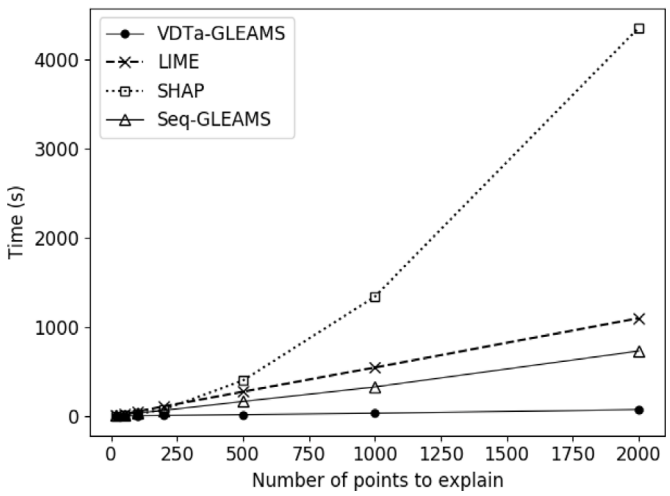

Fig. 14. Execution time of parallel and serial GLEAMS compared to that of different explainability techniques.

shown in Fig. 14 rather focus on the time to compute the explanations and highlight the superiority of our solution w.r.t. state-of-the-art approaches.

As mentioned before, GLEAMS yields a local and global explanation out-of-the-box, which translates into limited sensitivity to the number of points to explain, computationally wise. Nonetheless, Seq-GLEAMS, like LIME and SHAP, display a supposedly linear increase in computation time, w.r.t. the number of instances. The parallel implementation sensibly contributes to increasing the gap between GLEAMS and the other techniques.

6. Conclusion and future work

The explainability research field stems from the observation that ML algorithms have achieved impressive prediction accuracy, but their intrinsical structure often hinders the possibility of providing an explanation for the output. Various explainability algorithms have been proposed in the literature, each one with its own advantages and limitations. In this work, we focus on GLEAMS algorithm for its desirable feature of being able to provide both global and local explanations, and we deepen the various possibilities to parallelize its decision tree-based structure in order to reduce the computation time and environmental footprint. We chose Ray distributed framework to easily support the parallel execution and we propose four different approaches stemming from a detailed literature review of the state of the art for decision tree algorithms parallelization. An extensive evaluation of the proposed methods from the points of view of scalability, energy consumption and comparison with other explainability algorithms, which reveals the advantages and limitations of each version, also helps to highlight the best parallelization approach for GLEAMS.

The analysis of GLEAMS' code highlighted how the most computationally intensive part of the algorithm, i.e., the calculation of the best-split point, is inherently sequential. In the future, we plan to further investigate GLEAMS' parallelization opportunities, by deepening, for example, the possibility of changing the mathematical formula of the best-split point, so that also its computation can be distributed on various processors, allowing to reach a higher degree of maximum parallelism.

To the best of our knowledge, Ray has never been used to parallelize LIME or SHAP algorithms. Some recent practical attempts[10] involving

[10] https://www.databricks.com/blog/2022/02/02/scaling-shap-calculations-with-pyspark-and-pandas-udf.html; https://github.com/Affirm/shparkley; https://github.com/tinluu/LIME-on-Spark

Apache Spark exist instead. Also, a GPU-based parallelization has been proposed for TreeSHAP [72]. In the future, it would be interesting to compare the performance of our GLEAMS-Ray solution with these approaches.

Concerning our Ray implementation, in the future, the structure of the most promising solution VDTa-GLEAMS could be changed to parallelize the `async loop` itself. So, instead of having a single process handling events, a pool of executors would handle the incoming events in parallel. Currently, this solution is not possible because of Ray's limitations, and the use of Python multiprocessing is advised against in conjunction with Ray.[11] Once this restriction is overcome in Ray, VDTa-GLEAMS can be modified accordingly.

Another technical improvement could involve the employment of Ray's chained remote functions. Being able to accept both normal objects and object references as inputs, this kind of functions could, in principle, boost the performance of the VDT-GLEAMS parallel approach, by avoiding the use of any blocking `ray.get()` call.

Finally, various attempts to implement decision tree algorithms on GPUs have been proposed in literature [73–75]. In particular, we believe that taking inspiration from the general ideas exposed in [75], the use of GPU computing for the acceleration of the task-parallel part of GLEAMS' computation can represent a feasible matter of future work.

CRediT authorship contribution statement

Daniela Loreti: Writing – review & editing, Writing – original draft, Software, Methodology, Conceptualization. **Giorgio Visani:** Writing – review & editing, Writing – original draft, Software, Investigation, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The link to the open source code of our implementation is included in the Manuscript.

Acknowledgments

We thank Alessio Ferretti and Luca Ghedini (CCIB, CESIA, University of Bologna) for the precious assistance in the setup of the cluster evaluation environment; Gabriel Cortesi and Vincenzo Stanzione (M.Sc. in Computer Engineering at the University of Bologna) for the help in developing parts of the sequential and parallel versions of GLEAMS. This work has been realized by Daniela Loreti with a research contract co-financed by the European Union - PON Ricerca e Innovazione 2014–2020 ai sensi dell'art. 24, comma 3, lett. a), della Legge 30 dicembre 2010, n. 240 e s.m.i. e del D.M. 10 agosto 2021 n. 1062.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.future.2024.04.044.

[11] https://discuss.ray.io/t/is-it-possible-to-use-ray-in-a-subprocess-created-with-multiprocessing-process/7966.

References

[1] J. Kingston, Using artificial intelligence to support compliance with the general data protection regulation, Artif. Intell. Law 25 (4) (2017) 429–443, http://dx.doi.org/10.1007/s10506-017-9206-9.
[2] R. Guidotti, A. Monreale, S. Ruggieri, F. Turini, F. Giannotti, D. Pedreschi, A survey of methods for explaining black box models, ACM Comput. Surv. 51 (5) (2018) 93.
[3] S.M. Lundberg, S.-I. Lee, A unified approach to interpreting model predictions, Adv. Neural Inf. Process. Syst. 30 (2017).
[4] M.T. Ribeiro, S. Singh, C. Guestrin, Why should I trust you?: Explaining the predictions of any classifier, in: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2016, pp. 1135–1144.
[5] M. Craven, J.W. Shavlik, Extracting tree-structured representations of trained networks, Adv. Neural Inf. Process. Syst. (1996) 24–30.
[6] G. Visani, Meaningful Insights: Explainability Techniques for Black-Box Models on Tabular Data (Ph.D. thesis), University of Bologna, Italy, 2023, URL: http://amsdottorato.unibo.it/10934/1/PhD_Thesis.pdf.
[7] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M.I. Jordan, I. Stoica, Ray: A distributed framework for emerging AI applications, in: OSDI, USENIX Association, 2018, pp. 561–577.
[8] G. Visani, E. Bagli, F. Chesani, A. Poluzzi, D. Capuzzo, Statistical stability indices for LIME: Obtaining reliable explanations for machine learning models, J. Oper. Res. Soc. 73 (1) (2022) 91–101, http://dx.doi.org/10.1080/01605682.2020.1865846.
[9] G. Visani, E. Bagli, F. Chesani, Optilime: Optimized LIME explanations for diagnostic computer algorithms, in: S. Conrad, I. Tiddi (Eds.), Proceedings of the CIKM 2020 Workshops Co-Located with 29th ACM International Conference on Information and Knowledge Management, CIKM 2020, Galway, Ireland, October 19–23, 2020, in: CEUR Workshop Proceedings, vol. 2699, CEUR-WS.org, 2020, URL: https://ceur-ws.org/Vol-2699/paper03.pdf.
[10] V.M. Stanzione, Developing a New Approach for Machine Learning Explainability combining Local and Global Model-Agnostic Approaches (Master thesis), University of Bologna, Informatics and Engineering Department, 2022, URL: https://amslaurea.unibo.it/25480/1/msc_thesis.pdf.
[11] J.R. Quinlan, C4.5: Programs for Machine Learning, Elsevier, 1993.
[12] R.D. Gibbons, G. Hooker, M.D. Finkelman, D.J. Weiss, P.A. Pilkonis, E. Frank, T. Moore, D.J. Kupfer, The CAD-MDD: A computerized adaptive diagnostic screening tool for depression, J. Clin. Psychiatry 74 (7) (2013) 669.
[13] Y. Zhou, G. Hooker, Interpreting Models via Single Tree Approximation, 2016, arXiv: Methodology, URL: https://api.semanticscholar.org/CorpusID:88515329.
[14] J. Lei, M. G'Sell, A. Rinaldo, R.J. Tibshirani, L. Wasserman, Distribution-free predictive inference for regression, J. Amer. Statist. Assoc. 113 (523) (2018) 1094–1111.
[15] J.H. Friedman, Greedy function approximation: a gradient boosting machine, Ann. Statist. (2001) 1189–1232.
[16] G.J. Katuwal, R. Chen, Machine learning model interpretability for precision medicine, 2016, arXiv preprint arXiv:1610.09045.
[17] A.Y. Zhang, S.S.W. Lam, N. Liu, Y. Pang, L.L. Chan, P.H. Tang, Development of a radiology decision support system for the classification of MRI brain scans, in: 5th IEEE/ACM International Conference on Big Data Computing Applications and Technologies, BDCAT 2018, Zurich, Switzerland, December 17–20, 2018, IEEE Computer Society, 2018, pp. 107–115, http://dx.doi.org/10.1109/BDCAT.2018.00021.
[18] C. Moreira, R. Sindhgatta, C. Ouyang, P. Bruza, A. Wichert, An investigation of interpretability techniques for deep learning in predictive process analytics, 2022, arXiv preprint arXiv:2002.09192.
[19] D. Alvarez-Melis, T.S. Jaakkola, On the robustness of interpretability methods, 2018, arXiv preprint arXiv:1806.08049.
[20] M.T. Ribeiro, S. Singh, C. Guestrin, Anchors: High-precision model-agnostic explanations, in: AAAI, AAAI Press, 2018, pp. 1527–1535.
[21] M. Setzu, R. Guidotti, A. Monreale, F. Turini, D. Pedreschi, F. Giannotti, Glocalx-from local to global explanations of black box AI models, Artificial Intelligence 294 (2021) 103457.
[22] F. Harder, M. Bauer, M. Park, Interpretable and differentially private predictions, in: AAAI, AAAI Press, 2020, pp. 4083–4090.
[23] I. Sobol, Points which uniformly fill a multidimensional cube, Math. Cybern. Ser. (1985) 32.
[24] A. Zeileis, T. Hothorn, K. Hornik, Model-based recursive partitioning, J. Comput. Graph. Statist. 17 (2) (2008) 492–514.
[25] T. Chen, C. Guestrin, XGBoost: A scalable tree boosting system, in: B. Krishnapuram, M. Shah, A.J. Smola, C.C. Aggarwal, D. Shen, R. Rastogi (Eds.), Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13–17, 2016, ACM, 2016, pp. 785–794.
[26] S.R. Upadhyaya, Parallel approaches to machine learning - A comprehensive survey, J. Parallel Distrib. Comput. 73 (3) (2013) 284–292.
[27] N. Amado, J. Gama, F.M.A. Silva, Parallel implementation of decision tree learning algorithms, in: EPIA, in: Lecture Notes in Computer Science, vol. 2258, Springer, 2001, pp. 6–13.

[28] J. Chattratichat, J. Darlington, M. Ghanem, Y. Guo, H.F. Hüning, M. Köhler, J. Sutiwaraphun, H.W. To, D. Yang, Large scale data mining: Challenges and responses, in: KDD, AAAI Press, 1997, pp. 143–146.
[29] J.R. Quinlan, C4.5: Programs for Machine Learning, Morgan Kaufmann, 1993.
[30] R.A. Pearson, Chapter 17 - A coarse grained parallel induction heuristic, in: H. Kitano, V. Kumar, C.B. Suttner (Eds.), Parallel Processing for Artificial Intelligence, in: Machine Intelligence and Pattern Recognition, vol. 15, North-Holland, 1994, pp. 207–226, http://dx.doi.org/10.1016/B978-0-444-81837-9.50021-X.
[31] R. Kufrin, Decision trees on parallel processors, in: Parallel Processing for Artificial Intelligence 3, in: Machine Intelligence and Pattern Recognition, vol. 20, Elsevier, 1997, pp. 279–306.
[32] J.C. Shafer, R. Agrawal, M. Mehta, SPRINT: A scalable parallel classifier for data mining, in: VLDB, Morgan Kaufmann, 1996, pp. 544–555.
[33] M. Mehta, R. Agrawal, J. Rissanen, SLIQ: A fast scalable classifier for data mining, in: EDBT, in: Lecture Notes in Computer Science, vol. 1057, Springer, 1996, pp. 18–32.
[34] M.V. Joshi, G. Karypis, V. Kumar, ScalParC: A new scalable and efficient parallel classification algorithm for mining large datasets, in: IPPS/SPDP, IEEE Computer Society, 1998, pp. 573–579.
[35] R. Jin, G. Agrawal, Communication and memory efficient parallel decision tree construction, in: SDM, SIAM, 2003, pp. 119–129.
[36] Y. Ben-Haim, E. Tom-Tov, A streaming parallel decision tree algorithm, J. Mach. Learn. Res. 11 (2010) 849–872.
[37] K.W. Bowyer, L.O. Hall, T. Moore, N.V. Chawla, W.P. Kegelmeyer, A parallel decision tree builder for mining very large visualization datasets, in: SMC, IEEE, 2000, pp. 1888–1893.
[38] E. Bauer, R. Kohavi, An empirical comparison of voting classification algorithms: Bagging, boosting, and variants, Mach. Learn. 36 (1–2) (1999) 105–139.
[39] L. Breiman, Bagging predictors, Mach. Learn. 24 (2) (1996) 123–140.
[40] Q. Meng, G. Ke, T. Wang, W. Chen, Q. Ye, Z. Ma, T. Liu, A communication-efficient parallel algorithm for decision tree, in: NIPS, 2016, pp. 1271–1279.
[41] C. Fan, P. Li, Classification acceleration via merging decision trees, in: FODS, ACM, 2020, pp. 13–22.
[42] A.I. Weinberg, M. Last, Selecting a representative decision tree from an ensemble of decision-tree models for fast big data classification, J. Big Data 6 (2019) 23.
[43] D. Loreti, M. Lippi, P. Torroni, Parallelizing machine learning as a service for the end-user, Future Gener. Comput. Syst. 105 (2020) 275–286.
[44] W.M.P. van der Aalst, Distributed process discovery and conformance checking, in: FASE, in: Lecture Notes in Computer Science, vol. 7212, Springer, 2012, pp. 1–25.
[45] D. Loreti, F. Chesani, A. Ciampolini, P. Mello, Distributed compliance monitoring of business processes over MapReduce architectures, in: ICPE Companion, ACM, 2017, pp. 79–84.
[46] D. Loreti, F. Chesani, A. Ciampolini, P. Mello, A distributed approach to compliance monitoring of business process event streams, Future Gener. Comput. Syst. 82 (2018) 104–118.
[47] D. Loreti, A. Ciampolini, A distributed self-balancing policy for virtual machine management in cloud datacenters, in: HPCS, IEEE, 2014, pp. 391–398.
[48] A. Srivastava, E. Han, V. Kumar, V. Singh, Parallel formulations of decision-tree classification algorithms, Data Min. Knowl. Discov. 3 (3) (1999) 237–261.
[49] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[50] Y. Mu, X. Liu, Z. Yang, X. Liu, A parallel C4.5 decision tree algorithm based on MapReduce, Concurr. Comput. Pract. Exp. 29 (8) (2017).
[51] Y. Mu, X. Liu, L. Wang, A Pearson's correlation coefficient based decision tree and its parallel implementation, Inform. Sci. 435 (2018) 40–58.
[52] W. Dai, W. Ji, A MapReduce implementation of C4.5 decision tree algorithm, Int. J. Database Theory Appl. 7 (1) (2014) 49–60.
[53] B. Panda, J. Herbach, S. Basu, R.J. Bayardo, PLANET: Massively parallel learning of tree ensembles with MapReduce, Proc. VLDB Endow. 2 (2) (2009) 1426–1437.
[54] S. Samsani, A comparative analysis on parallel implementations of decision tree learning for large scale complex datasets in apache spark, Int. J. Creat. Res. Thoughts 9 (5) (2021) 248–255.
[55] A. Segatori, F. Marcelloni, W. Pedrycz, On distributed fuzzy decision trees for big data, IEEE Trans. Fuzzy Syst. 26 (1) (2018) 174–192.
[56] Y. Mu, X. Liu, L. Wang, J. Zhou, A parallel fuzzy rule-base based decision tree in the framework of map-reduce, Pattern Recognit. 103 (2020) 107326.
[57] A. Nasridinov, Y. Lee, Y. Park, Decision tree construction on GPU: ubiquitous parallel computing approach, Computing 96 (5) (2014) 403–413, http://dx.doi.org/10.1007/S00607-013-0343-Z.
[58] D. Strnad, A. Nerat, Parallel construction of classification trees on a GPU, Concurr. Comput. Pract. Exp. 28 (5) (2016) 1417–1436, http://dx.doi.org/10.1002/CPE.3660.
[59] K. Jurczuk, M. Czajkowski, M. Kretowski, Evolutionary induction of a decision tree for large-scale data: a GPU-based approach, Soft Comput. 21 (24) (2017) 7363–7379, http://dx.doi.org/10.1007/S00500-016-2280-1.
[60] K. Jurczuk, M. Czajkowski, M. Kretowski, GPU-accelerated evolutionary induction of regression trees, in: C. Martín-Vide, R. Neruda, M.A. Vega-Rodríguez (Eds.), Theory and Practice of Natural Computing - 6th International Conference, TPNC 2017, Prague, Czech Republic, December 18–20, 2017, Proceedings, in: Lecture Notes in Computer Science, vol. 10687, Springer, 2017, pp. 87–99, http://dx.doi.org/10.1007/978-3-319-71069-3_7.
[61] K. Jurczuk, M. Czajkowski, M. Kretowski, Fitness evaluation reuse for accelerating GPU-based evolutionary induction of decision trees, Int. J. High Perform. Comput. Appl. 35 (1) (2021) http://dx.doi.org/10.1177/1094342020957393.
[62] K. Jurczuk, M. Czajkowski, M. Kretowski, GPU-based acceleration of evolutionary induction of model trees, Appl. Soft Comput. 119 (2022) 108503, http://dx.doi.org/10.1016/J.ASOC.2022.108503.
[63] K. Jurczuk, M. Czajkowski, M. Kretowski, Adaptive in-memory representation of decision trees for GPU-accelerated evolutionary induction, Future Gener. Comput. Syst. 153 (2024) 419–430, http://dx.doi.org/10.1016/j.future.2023.12.003.
[64] K. Jurczuk, M. Czajkowski, M. Kretowski, Multi-GPU approach to global induction of classification trees for large-scale data mining, Appl. Intell. 51 (8) (2021) 5683–5700, http://dx.doi.org/10.1007/S10489-020-01952-5.
[65] V.G. Costa, S. Salcedo-Sanz, C.E. Pedreira, Efficient evolution of decision trees via fully matrix-based fitness evaluation, Appl. Soft Comput. 150 (2024) 111045, http://dx.doi.org/10.1016/j.asoc.2023.111045, URL: https://www.sciencedirect.com/science/article/pii/S1568494623010633.
[66] F.S. Luan, S. Wang, S. Yagati, S. Kim, K. Lien, I. Ong, T. Hong, S. Cho, E. Liang, I. Stoica, Exoshuffle: An extensible shuffle architecture, in: SIGCOMM, ACM, 2023, pp. 564–577.
[67] S. Zhuang, Z. Li, D. Zhuo, S. Wang, E. Liang, R. Nishihara, P. Moritz, I. Stoica, Hoplite: efficient and fault-tolerant collective communication for task-based distributed systems, in: SIGCOMM, ACM, 2021, pp. 641–656.
[68] G. Cortesi, Design, Implementation and Evaluation of Parallel Solutions for a Nested Explainability Algorithm (Master thesis), University of Bologna, Informatics and Engineering Department, 2023.
[69] J.H. Friedman, Multivariate adaptive regression splines, Ann. Statist. 19 (1) (1991) 1–67.
[70] L. Breiman, Bagging predictors, Mach. Learn. 24 (1996) 123–140.
[71] M.D. Hill, What is scalability? SIGARCH Comput. Archit. News 18 (4) (1990) 18–21.
[72] R. Mitchell, E. Frank, G. Holmes, GPUTreeShap: massively parallel exact calculation of SHAP scores for tree ensembles, PeerJ Comput. Sci. 8 (2022) e880, http://dx.doi.org/10.7717/PEERJ-CS.880.
[73] N. Pilkington, H. Zen, An implementation of decision tree-based context clustering on graphics processing units, in: INTERSPEECH, ISCA, 2010, pp. 833–836.
[74] T. Sharp, Implementing decision trees and forests on a GPU, in: ECCV (4), in: Lecture Notes in Computer Science, vol. 5305, Springer, 2008, pp. 595–608.
[75] B. Ren, S. Balakrishna, Y. Jo, S. Krishnamoorthy, K. Agrawal, M. Kulkarni, Extracting SIMD parallelism from recursive task-parallel programs, ACM Trans. Parallel Comput. 6 (4) (2019) 24:1–24:37.

Daniela Loreti is junior assistant professor of Operating Systems at Department of Computer Science and Engineering, University of Bologna. She received her Ph.D. in Computer Science in 2016. Her research focuses on distributed systems for big data management and stream processing as well as parallel paradigms for high performance computing. She is also interested in the parallelization of artificial intelligence techniques in the fields of machine learning, process mining and expert systems.

Giorgio Visani currently works as a Researcher in the Artificial Intelligence group at University of Bologna. After B.Sc. and M.Sc. in Statistics, he took on Ph.D. in Data Science in the hometown university. He has been visiting researcher at Universitè Libre de Bruxelles and University of Sydney, with whom he has active collaborations. His research interests lie in Explainable Machine Learning, Causal Inference and blending together Causality with Machine Learning, Data Generation and Assessment of their quality.