Performance Analysis of Decision Tree Learning Algorithms on Multicore CPUs

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 156

Systems Group, Department of Computer Science, ETH Zurich

in collaboration with

Oracle Labs

Performance Analysis of Decision Tree Learning Algorithms on Multicore CPUs

by

Jingyi Wang

Supervised by

Prof. Dr. Gustavo Alonso
Dr. Ingo Müller
Dr. Davide Bartolini

December 1, 2016

Informatik
Computer Science

Abstract

Decision trees are widely-used in data mining and machine learning. They are easy to interpret and are key components of ensemble methods like random forests and gradient boosting decision trees. The main tasks of growing a decision tree include: scanning the data at each tree node, computing a class frequency-based metric to find the best splitting point, and splitting the node into child nodes. As multicore CPUs are commonly used nowadays, we investigate in this thesis various ways to parallelize the C4.5 decision tree algorithm to improve its running speed on multicore CPUs.

One challenge in parallelizing the growing of a decision tree is the difficulty to achieve efficient memory bandwidth utilization throughout the whole algorithm. This is due to the need to sort numerical attributes (i.e. attributes that potentially have continuous numerical values) in order to find the splitting point for each node. Another challenge is that tree nodes of different size favor different types of parallelism, which implies that the scheme of parallelism should be able to adapt to the growing of the tree dynamically to achieve high performance. Our goal is to address theses two problems in decision tree parallelizing.

We start from a common approach, which pre-sorts numerical attributes once at the beginning (we call these algorithms sort-based algorithms). We study multiple ways to parallelize the sort-based algorithm and found that keeping numerical attributes sorted is expensive because different attributes have different sortings and this makes the splitting costly. To avoid pre-sorting, we implement a count-based algorithm that eliminates the need for keeping attributes sorted and makes the splitting cheaper. We parallelize the sequential count-based algorithm and use a highly tuned partitioning routine to fully use available memory bandwidth. Our experiments show that the use of partitioning leads to shorter execution time and better scalability.

On the high level, there are three types of parallelism that could potentially be employed: attribute-level, record-level and node-level parallelism. Through the growing of a tree, there are situations where only one of them is available. We hence need to express all of the three types of parallelism as much as possible. Thus, we further take advantage of a task parallelism scheme that utilizes a work stealing strategy to reduce barriers and employs different types of parallelism on the fly. This results in our fastest parallel algorithm, which achieves 10x speedup on a 16-core machine.

Acknowledgements

First and foremost, I would like to express my great appreciation to my supervisors Dr. Ingo Müller and Dr. Davide Bartolini, for their supervision and support throughout my work. They helped me get started with this area and provided me with a lot of advices both theoretically and practically. This thesis would not have been possible without their guidance. I would also like to thank Prof. Gustavo Alonso for sharing his knowledge and giving many helpful suggestions at the crucial moments of the development of this thesis. Furthermore, I would like to thank my fellow lab mates in the group for stimulating discussions. Last but not least, I would like to thank my family for their constant love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Decision trees have many applications, from statistics to data mining and machine learning. This has been a field of many researches and applications. Microsofts motion controller Kinect [20], for example, uses a randomized decision tree bagging learned from over 1 million training examples to infer body parts. It is also common for biological studies like [18] to conduct analyses using decision tree models built from large datasets.

A decision tree (Figure 1.1) classifies data records based on their attribute values. Each non-leaf node represents a test on an attribute (e.g. whether the weather is sunny or not). Each leaf node represents a class label. When classifying, each data record walks through the tree from the root to a leaf according to the outcome of each test and is assigned to a class label in the end. Training a decision tree requires scanning and evaluating all the data at each node. If the training dataset is large, training a decision tree classifier from it could take a lot of time.

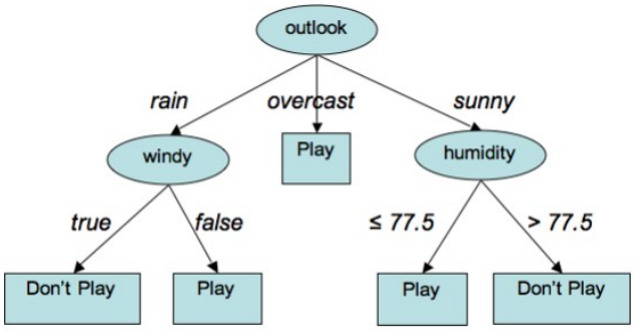


Figure 1.1: An example of a decision tree. Given certain weather conditions, this tree is used to predict whether a person would play golf or not.

It is therefore important to go beyond training decision trees with high accuracy, but also to train at a high speed. Since parallel hardware architectures are embraced by industry nowadays, quite a few efforts have been spent on training a decision tree by employing parallel approaches. Some perform parallelization techniques directly on the data [32, 22]. Some exploit the use of compact representations of the original data to accelerate the processing of numerical attributes [11, 14]. Others [34, 31], map the sections of the tree induction algorithm to a hardware architecture like GPU and FPGA.

A challenge of parallelizing decision tree algorithms is the way of handling numerical attributes, which is a key factor that determines the tree growing speed to a great extent. In order to find the splitting point for each node, numerical attributes have to be sorted so that we can sequentially scan the attributes and for each candidate splitting point, we compute the metric that is based on class frequency of records whose values are above the candidate splitting point. RainForest [11] uses compact representations to reduce the amount of data that needs to be sorted. Based on it, SPIES [14] parallelizes the growing of large nodes and further reduce data size by sampling. Some other algorithms uses techniques to accelerate the sorting itself, like [29] and [34].

Another challenge comes from the irregular structure of a tree. At top levels of the tree, data parallelism is preferred since there are just a few large nodes. As the tree grows, node-level parallelism becomes important, as it allows concurrent growing of small nodes. Thus, different types of parallelism are preferred in different parts of the tree. Amado et al. [3] proposed a hybrid algorithm which grows large nodes using data parallelism and for each small node, it lets a processor continue alone the construction of subtree rooted at the node. Aldinucci et al. [2] apply attribute-level parallelism on large nodes and node-level parallelism on small nodes dynamically, but they do not utilize record-level parallelism.

In this thesis, we explore different ways to parallelize the classic C4.5 decision tree algorithm on multicore CPUs, with the aim of achieving high growing speed without using any additional mathematical simplifications (e.g. estimators, sampling). We first investigate multiple parallelization methods of the sort-based algorithm. We then take advantage of the compact representations, known as the AVC-sets of numerical attributes [32] as well as a highly tuned partitioning routine [23] to achieve memory bandwidth efficiency. We also use task parallelism scheme [8] to reduce barriers and to allow dynamic employment of different types of parallelism. This algorithm is shown by our experimental results as the fastest parallel algorithm and achieves 10x speedup on sixteen cores, which is about 2 to 3 times faster than the parallel sort-based algorithm.

The rest of the thesis is organized as follows: In Chapter 2, we review the background and related work of the decision tree algorithms and its existing parallelization methods. In Chapter 3, we study and evaluate the performance of the sort-based tree induction algorithms. Chapter 4 describes the count-based algorithms as well as the use of the task parallelism scheme. After that, we give a performance evaluation of the count-based algorithms and compare them with the sort-based algorithms. We conclude with a summary in Chapter 5.

Chapter 2

Background and Related Work

In this chapter we cover relevant aspects of decision tree induction and the existing sequential and parallel approaches.

2.1 Decision Tree

Decision trees have been widely used for classification problems. When classifying a data records, it traverses the tree and at each node, makes decision on to which branch it is going based on its attribute values until it finally arrives at a leaf node and is assigned the corresponding class label. A decision tree consists of three elements: splitting nodes, branches, and leaves. The input data is usually stored in a relational database with rid (record id), attributes, and class as fields. In the rest of the thesis, we will use the word record to refer to a row in the database. A splitting node represents a test on one or several attributes, and a branch represents the outcome of the test.

Decision trees can also be used for regression problems. Regression trees are very similar to classification trees but the dependent variable takes continuous values. The predicted outputs of a regression tree are often the average values of the records at the leaf nodes.

2.2 Training a Decision Tree

To grow a decision tree, one starts from the root and scans the data to find an attribute value that, if used to split that node, gives the best score of some metric. This procedure is then recursively applied to each of the child nodes. Commonly used metrics include information gain, gain ratio and the Gini impurity (Definition 1-3). They are all based on the calculation of the frequency of each class at a node. Given a splitting node, we use S to denote the dataset of that node and f_i to

denote the frequency of class i. A is the splitting attribute and m is the number of different classes.

Definition 1. *Information gain*

$$InfoGain(S, A) = H(S) - \sum_{v \in Value(A)} \frac{|S_v|}{|S|} H(S_v)$$

$$H(S) = Entropy(S) = -\sum_{i=1}^{m} f_i \log f_i$$

Definition 2. *Gain ratio*

$$GainRation(S, A) = \frac{InfoGain(S, A)}{SplitEntropy(S, A)}$$

$$SplitEntropy(S, A) = -\sum_{v \in Value(A)} \frac{|S_v|}{|S|} \log \frac{|S_v|}{|S|}$$

Definition 3. *Gini impurity*

$$I_G(f) = \sum_{1}^{m} f_i(1 - f_i)$$

Once a tree is generated, pruning techniques are sometimes applied to avoid the problem of overfitting. There are numerous tree pruning techniques, some of them are: reduced error pruning (used by C4.5), cost complexity pruning (used by CART), and minimum description length (used by SLIQ). The pruning phase is a well-studied problem and in most cases only takes a very small amount of the entire training time. Moreover, if the tree is built as a part of an ensemble model, the pruning phase is not always needed.

Records with missing values are typically not used in the induction of a tree. In the classification phase, the following approaches are commonly used to distribute these records to child nodes:

- Distributing them to all child nodes, but with diminished weights, which are proportional to the number of records from each child node.
- Use surrogates to distribute records to a child node, where surrogates are input features which resembles best how the test feature send data records to left or right child node.
- All the records go to the node which already has the most records.
- Distributing them randomly to only one single child node (often used for a faster running time).

2.3 Related Work

This section introduces related work of decision tree algorithms.

2.3.1 Sequential Algorithms

ID3 The ID3 algorithm [26] only handles categorical attributes and uses a divide and conquer method to construct a tree recursively:

1) Begin with the original set S as the root node, check if the node is a leaf.

2) Calculate the entropy of every attribute using the data set S.

3) Split the set $\mathcal{S}$ into subsets using the attribute A for which entropy is minimum.

4) For each value of A, create a new child node.

5) Recurse on child nodes using remaining attributes.

C4.5 The C4.5 algorithm [27] is an extension of the ID3 algorithm. The main improvement is that it handles both categorical and numerical attributes. Categorical attributes are processed using the same method as in ID3, numerical attributes are sorted and each possible split point (the smallest value of each interval $\{v_i, v_{i+1}\}$, where v_i and v_{i+1} are two adjacent different values of an attribute) is evaluated. Our work is based on this algorithm.

CART Classification and Regression Trees (CART) [16] processes numerical attributes the same way as C4.5 does. For categorical attributes, it considers every possible binary grouping of the attribute values.

SLIQ Both the C4.5 and the CART algorithm require repeated sorting of numerical attributes at each node of the tree, which is very expensive and not scalable for large dataset. Mehta et al. [22] proposed an algorithm called SLIQ that addresses this problem by performing pre-sorting at the beginning of the training phase. It then tracks the position of each record using a class list and grows the tree in a breadth-first manner (Figure 2.1). Although it outperforms C4 (the predecessor of C4.5) and CART, its scalability is limited by the size of the training set. Once the training set gets large, the class list can no longer fit in memory, which degrades the performance since the class list is frequently accessed and updated.

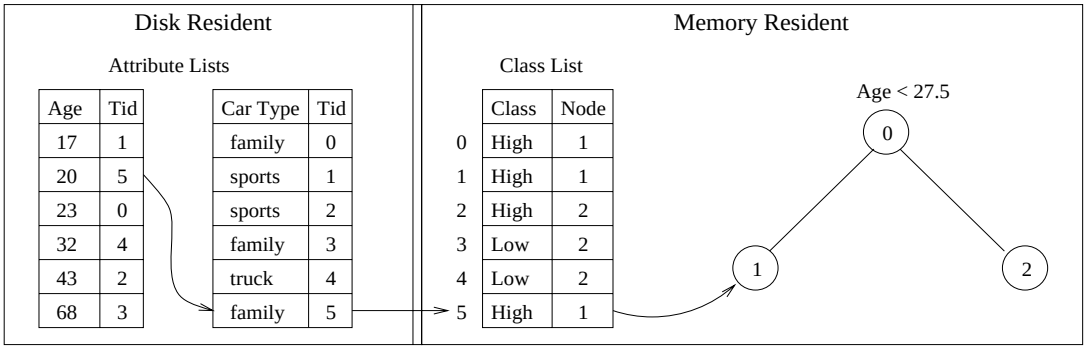


Figure 2.1: Structure of SLIQ

SPRINT Shafer et al. [32] proposed an algorithm called SPRINT which removes the memory restrictions of decision tree learning. It also uses strategies like pre-sorting and breadth-first growing like the SLIQ does, but the attribute lists are constructed in a way that each of them contains a class column (Figure 2.2), which makes it easier to scale to large dataset. However, the attribute lists need to be rewritten each time when splitting a node (Figure 2.3) and this takes more time. The authors showed that serial SPRINT has better performance than SLIQ when the input size is large enough, and it is also very easy to parallelize. We use a similar technique for the sort-based algorithms in Section 3.1

Table 2.1 summarizes the major sequential tree learning algorithms.

Age	Class	rid
17	High	1
20	High	5
23	High	0
32	Low	4
43	High	2
68	Low	3

Car Type	Class	rid
family	High	0
sports	High	1
sports	High	2
family	Low	3
truck	Low	4
family	High	5

Figure 2.2: SPRINT-Attribute lists

Other methods Other notable serial algorithms include: (1) RainForest [11], a framework that avoids the out-of-memory sorting of all the numerical attributes by summarizing the dataset into the so-called AVC (Attribute value, Class label) groups, which usually fit in memory. During the tree growing phase, only in-memory sort of AVC groups needs to be performed repeatedly. The

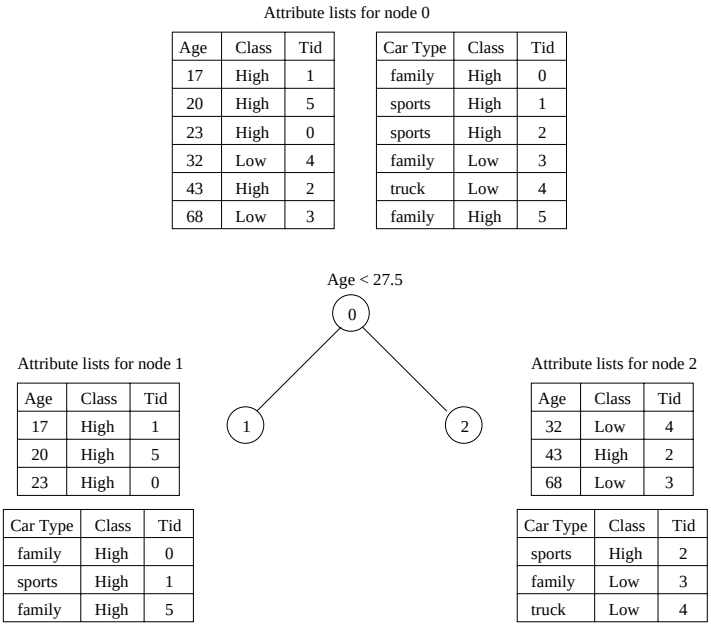


Figure 2.3: SPRINT-splitting the attribute lists of a node

Table 2.1: Comparison of some classic sequential tree learning algorithms

Name	Attribute type	Splitting metric	Splitting pattern	Growing strategy	Pruning
ID3	categorical	information gain	multiple	depth-first	no
C4.5	numerical/categorical	gain ratio	multiple	depth-first	yes
CART	numerical/categorical	gini impurity	binary	depth-first	yes
SLIQ	numerical/categorical	gini impurity	binary	breadth-first	yes
SPRINT	numerical/categorical	gini impurity	binary	breadth-first	yes

authors showed that these in-memory sorts are ten times faster than the initial creation of attribute lists in SPRINT. We also take advantage of the AVC groups for our AVC-based algorithms in Section 4.1, (2) CLOUDS [28] reduces computation and I/O complexity by sampling the splitting points for numerical attributes and using an estimation step to further narrow the candidate space of the best split point, (3) BOAT [10] combines decision tree learning with bootstrap sampling to yield a faster and accurate solution. It constructs an initial tree using small subsets of the training dataset and then refine it to the final tree, (4) SURPASS [17] summarizes numerical attributes into a set of statistics (mean vectors and covariance matrices) and uses it to generate splitting criteria. These statistical information can be gathered incrementally from the data, thus the memory bound is avoided. The author compared it to RainForest and showed that it achieves competitive accuracy

and lower CPU time.

2.3.2 Parallel Algorithms

A variety of parallel decision tree learning algorithms have been proposed.

Parallel SPRINT The parallel version of SPRINT [32] partitions the data horizontally and gives each processor a part of each attribute list (Figure 2.4). For numeric attributes, each processor scans the sub lists it receives and calculates its own best split point for the node. All the processors then send the best split points to a coordinator to find the overall best split point. For categorical attributes, each processor builds a count matrix for the node while scanning the sub lists. The coordinator collects all the count matrices and sums them to get a global count matrix, which is then used to find the best split point. After splitting on the splitting attribute of a node, we need to split the data of the remaining non-splitting attributes. In order to do this, all processors need to communicate with each other and a hash table needs to be stored on every one of them which stores the rid (i.e. record ID) and the corresponding node ID to keep track of which node each record goes to. This means that SPRINT is not scalable in run-time as well as memory requirements.

Later, ScalParc [15] was proposed to further extend the scalability of SPRINT by applying a distributed hash table. It uses attribute lists and finds the best split point the same way as SPRINT does. In addition, all processors together maintain a global node table which indicates to which child node each record goes. The node table is distributed among all processors such that each processor only needs to store a part of it. When performing splitting, each processor first uses a hash buffer to update its part of the node table. After the updating of all processors are done, they can use enquiry buffers to retrieve information from the updated node table. This distributed hashing scheme reduces communication cost.

Processor 0

Age	Class	rid
17	High	1
20	High	5
23	High	0

Car Type	Class	rid
family	High	0
sports	High	1
sports	High	2

Processor 1

Age	Class	rid
32	Low	4
43	High	2
68	Low	3

Car Type	Class	rid
family	Low	3
truck	Low	4
family	High	5

Figure 2.4: Data distribution in parallel SPRINT

Implementations on MapReduce Panda et al. [25] presented PLANET, a MapReduce framework for regression tree learning. Prior to the induction a MapReduce job is run on the dataset to compute approximate equidepth histograms for every numerical attribute. When evaluating splits on a numeric attribute, a single split point is considered from every histogram bucket of that attribute. Thus, splits are not evaluated between every pair of values of the attribute. To find the best split for a tree node, a MapReduce job is created and several such jobs are run in parallel. A mapper calculates the number and summation of the class (dependent) variables it receives. Each reducer then uses these information to calculate variance reduction and outputs what it considers to be the best split point of the node. In the end, a controller aggregates the split points from the reducers and chooses the best overall split for the node. In order to reduce the start-up cost of MapReduce, the authors introduced a forward scheduling trick. It uses the controller to run a background thread which keeps setting up MapReduce jobs. These jobs continuously wait for new node-split tasks from the scheduler. In order to reduce the tear-down cost, the controller runs a thread that periodically checks for the output files and processes them immediately.

A MapReduce-based implementation of C4.5 algorithm was described in [9] . It uses the same attribute list structure as the SPRINT does, and splits the dataset vertically before assigning the subsets equally among several nodes. The entropy of each attribute value is computed on a mapper and the reducers compute the gain ratio if splitting on that attribute value. This algorithm also maintains a hash table which records the mapping between the samples and the current nodes they arrive at. The algorithm grows the tree one node at a time, thus it does not have to let each record traverse through the current model each time when it splits a node, which also means that it loses the node-level parallelism.

There are some drawbacks of the MapReduce frameworks for decision tree learning: First, all these implementations require multiple MapReduce phases, which makes MapReduce start-up and tear-down costs the primary bottleneck of the systems performance. Although PLANET uses forward scheduling to mediate this problem, the forward scheduling of too many jobs can still result in waste of resources; Second, it is hard to achieve node-level parallelism while not having the entire training dataset traverse through the already-generated model.

Implementations on GPU Several GPU-based parallel constructions of individual decision trees have been reported. Sharp [33] first implemented the evaluating and training of decision trees and forests entirely on a GPU and presented its usage in the context of object recognition. The algorithm used common features in computer vision as input data and received a speedup of 8x in the decision tree training phase over a standard CPU implementation. Nasridinov et al. [24] exploited parallelsim of the ID3 algorithm for decision tree learning on GPU. The proposed solution grows the tree node-by-node recursively and allows for multi-way splits of a node. Parallel search for the best split point is combined with data partitioning on the GPU to achieve a speedup of 4x compared to the parallel Java open source project Weka [13] and lower energy consumption.

Chiu et al. [6] presented a CUDA implementation of a decision tree model based on the parallel

SPRINT algorithm. An extended version of it named CUDT was propsed by Lo et al. [19] , where the CPU is responsible for flow control and the GPU is responsible for computation. It processes one tree node at a time and performs parallel search for the best split point by using divided attribute lists. Parallel primitives such as parallel sorting and reduction were leveraged, but the authors did not describe specifically the mapping between work and GPU threads. The results show that CUDT is 5 to 5.5 times faster than Weka and 18 times faster than the sequential SPRINT for numerical datasets.

A more recent complement and extension of the above algorithms is presented by Strnad et al. [34]. It implements a CART-like decision tree learning model with binary splitting. It maintains a task priority queue which sorts the current tree nodes to be processed based on the number of records in each of them to achieve load balance and then concurrently processes a batch of tree nodes at a time. Each GPU block is responsible for one attribute-node pair as illustrated in the Figure 2.5, where thread block (i,j) evaluates all possible splits of the j-th attribute for the i-th node. Thus, it achieves node-level parallelism on the horizontal dimension and attribute level parallelism on the vertical dimension. Further, it also leverages some parallel primitives like prefix-sum within each block to achieve split-level parallelism. The authors demonstrated a speedup of around 10x when compared with a multi-threaded CPU implementation running on an Intel quad-core CPU. The experimental results also show that memory latency is the bottleneck in this design.

One problem of this algorithm is that for datasets with a small number of attributes, the GPU grid size may be too small at top levels of the tree to provide enough resident blocks per streaming multiprocessor to hide the memory latency well. Another problem appears as the tree construction progresses, the amount of attribute-level parallelism remains constant, but node-level parallelism grows and split-level parallelism diminishes. This leads to larger grids but smaller blocks, which causes lower occupancy of the GPU. These problems appear because the work assignment on the GPU is limited to certain structures (i.e. blocks and grids), which are highly parallelized but somehow are not so flexible in load balancing. The authors proposed possible solutions to address these two problems, which are to use multiple blocks operating on a single attribute at upper levels of the tree and processing several attributes or nodes within larger thread blocks at lower levels of the tree.

A comparison of some recent parallel decision tree algorithms is given in table 2.2.

Table 2.2: Comparison of some recent parallel tree learning algorithms

Name	Architecture	Predecessor	Attribute type	Splitting pattern	Node parallelism
PLANET	MapReduce	regression tree	numerical/categorical	binary	yes
Dai et al.	MapReduce	C4.5	numerical/categorical	multiple	no
Nasridinov et al.	GPU	ID3	categorical	multiple	no
CUDT	GPU	SPRINT	numerical	binary	no
Strnad et al.	GPU	C4.5	numerical/categorical	binary	yes

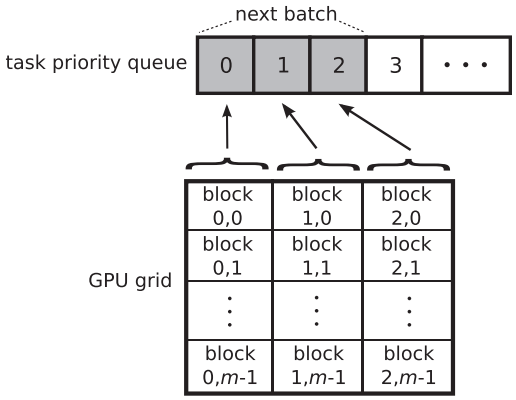


Figure 2.5: GPU Implementation of CART. Each GPU block is responsible for one attribute-node pair.

Other methods Other notable parallel decision tree learning algorithms include: (1) The SPIES algorithm [14] combines AVC groups from RainForest with sampling to achieve memory-efficient processing for numerical attributes. Zaki et al. [35] presented a parallel decision tree learning algorithm on shared-memory multiprocessors. The method grows the tree level by level and uses attribute scheduling. It is then extended with task pipelining and dynamic load balancing to yield speedup, but there are still certain barriers at each level of the tree; (2) Ben-Haim et al. [4] proposed another algorithm for the classification of large dataset and streaming data. The algorithm uses data parallelism and each processor quickly constructs a histogram which can be seen as an approximate representation of the data subset it sees. Later, a master gathers this information from the processors to find the best split points of a tree node; (3) Giannella et al. [12] presented an algorithm that minimizes the number of messages transmitted when constructing a tree over heterogeneously distributed (i.e. verticaly partitioned) data with only categorical attributes. It does so by approximating information gain using random projections and achieves 80% of the accuracy with 20% of the communication cost compared to centralized version.

2.3.3 Summary

The classic and widely-used C4.5 algorithm is the basis of many prior works. The key procedure of the algorithm is to decide how to split a node. In order to do this, all numerical attributes or some representations of the numerical attributes need to be sorted. Various algorithms have been proposed to handle this, which can be roughly grouped into three types of methods:

- Sorting the numerical attributes each time when growing a node. This is difficult to accommodate to large datasets and is rarely used now.

- Performing pre-sorting (sort-based algorithms) [32, 22]. All the attributes are sorted before the algorithm starts, and must be kept in order until the very end of the program. This requires data to be stored and kept in separate tables. Maintaining the sorted tables might generate additional cost.
- Constructing some condensed representations of the original data and then sorting the representations. This saves the time of pre-sorting but how to efficiently construct and sort the representations repeatedly is a challenge. The most renowned one of these structures is the AVC-group used by [11, 14]. We denote algorithms utilizing AVC-group as AVC-based algorithms.

While it is known that the first method takes the longest time among the three, it is not clear which of the other two performs better. In the next chapter, we explore the sort-based algorithms. Since there are already various existing parallel sorting algorithms, we will only focus on the tree growing part.

Chapter 3

Sort-based Decision Tree Induction

In this chapter, we present the sort-based decision tree induction algorithms that are explored in our work. Section 3.1 gives an overview of the basic sequential induction algorithm. Section 3.2 describes the parallelism types and tree growing patterns that are used in the project. Then we present in Section 3.3 how we parallelize the sequential algorithm by combining different parallelism types and growing patterns. The evaluation of performance is given in section 3.4.

3.1 Sequential Algorithm

Algorithm 3.0 demonstrates the basic C4.5-like sequential algorithm. Assuming our training dataset is the one as shown in Figure 3.1, we first store it in attribute lists (see Figure 3.2, we will take attribute Outlook and Temperature as examples). An attribute list has three fields: the record ID (rid), the attribute value and the class label. Record ID and class label are stored as integers, attribute value is stored as an integer (if the attribute is categorical) or a single-precision floating point (if the attribute is numerical). Each row in the attribute list corresponds to a record in the original table and numerical attribute lists are pre-sorted by value. The algorithm begins with an empty root node and attribute lists for the root. It first evaluates all the attribute lists and computes the information gain we can get if splitting on that attribute. In order to calcluate the value of f_i in the formula of information gain, we use an assistant data structure, count matrix (shown in Figure 3.3), the same way as the SPRINT algorithm does. For a categorical attribute, the count matrix stores frequencies of different class labels with respect to the attribute values. For a numerical attribute, we evaluate each candidate splitting point in that attribute list. A point is a candidate splitting point if it has a different value than its successor (except for the last point), so the count matrix stores class frequencies both above and below a certain numerical point. Thus, after a scan over an attribute list, we know the information gain of this attribute. If the attribute is a numerical one, we will also know the best splitting point of that attribute.

Algorithm 3.0 sequential

```
growTree (TreeNode n):
    if (n is a leaf):
        return
    end if

    for each attribute a ∈ A do:
        Evaluate a
    end for

    Find the best splitting attribute
    Update position array node_ids

    for each attribute a ∈ A do:
        Split a by node_ids
    end for
    for each child node c do:
        growTree(c)
    end for
```

rid	Outlook	Temperature	Humidity	Windy	toPlay
0	Sunny	85	85	False	Don't play
1	Sunny	80	90	True	Don't play
2	Overcast	80	78	False	Play
3	Rainy	70	96	False	Play
4	Rainy	68	80	False	Play
5	Sunny	80	88	True	Play

Figure 3.1: An example of the dataset for training a decision tree model.

After the evaluation, the attribute with the highest information gain is chosen to be the splitting attribute. If it is a categorical one, the attribute lists will be split according to the distinct values of that attribute and the attribute will not be used in further splits. If it is numerical, the tree performs a binary split and the attribute will still be considered in the evaluation of the child nodes. Since attribute lists are linked only by the rids, we need to keep an array of integers, $node\_ids$, which stores for each record, the ID of the child node to which the record should be going to. The

Outlook

rid	value	label
0	Sunny	0
1	Sunny	0
2	Overcast	1
3	Rainy	1
4	Rainy	1
5	Sunny	1

Temperature

rid	value	label
0	85	0
1	80	0
2	80	1
3	70	1
4	68	1
5	80	1

Sort

Temperature

rid	value	label
4	68	1
3	70	1
1	80	0
2	80	1
5	80	1
0	85	0

Figure 3.2: Attribute lists for the categorical attribute Outlook and the numerical attribute Temperature.

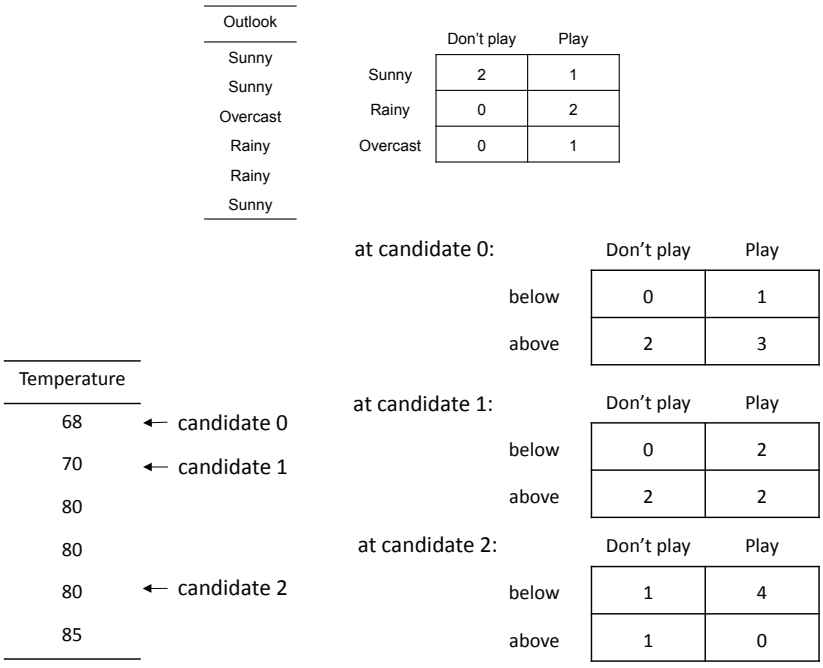
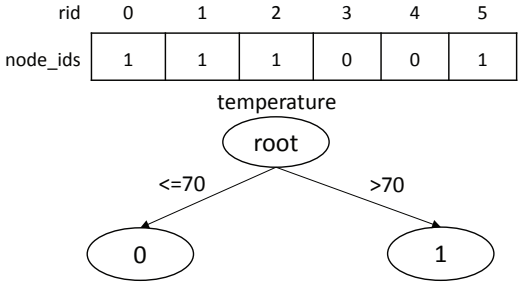


Figure 3.3: Use count matrices to evaluate categorical and numerical attributes.

$node\_ids$ is updated by going through the splitting attribute list once. If the splitting attribute is numerical, we compare each record's attribute value with the splitting value. If it is smaller or equal than it, the record goes to the left child, otherwise it goes to the right child. If the splitting attribute is categorical, we assign each record to a child according to its attribute value.

After that, we split the current attribute lists into sub-lists for the child nodes according to $node\_ids$ (Figure 3.4). We scan the the attribute lists, and for each record with record ID rid, we consult the

rid-th element in the $node\_ids$ array, which is the child ID for the record. We then send the record to the corresponding child node. When the splitting is finished, the growTree method is called recursively on each of the child nodes until all the nodes become leaves. A node becomes a leaf if all the records in that node belong to the same class or splitting that node gives a negative information gain. Sometimes pruning may be needed if the tree is not part of an ensemble algorithm, but it usually takes a very small amount of training time, so we will not focus on that aspect in this project.



Outlook

rid	value	label
3	Rainy	1
4	Rainy	1

Outlook

rid	value	label
0	Sunny	0
1	Sunny	0
2	Overcast	1
5	Sunny	1

Temperature

rid	value	label
4	68	1
3	70	1

Temperature

rid	value	label
1	80	0
2	80	1
5	80	1
0	85	0

Figure 3.4: Split attribute lists.

3.2 Parallelism Types and Tree Growing Patterns

In this section, we introduce concepts that are used later in the description of our parallel algorithms. Parallelism types correspond to the parallelization techniques applied in each node. Tree growing patterns refer to the growing patterns of the whole tree.

3.2.1 Attribute-level Parallelization

Attribute-level parallelism splits the attribute lists vertically and distribute them among threads (Figure 3.5). When evaluating attributes, each thread is dynamically assigned to one or more attribute lists and computes the information gain of its own attribute(s). After the evaluation, each thread should have the information gain of what it sees is the best splitting attribute. Then one thread gathers the information gains from all other threads and finds the global best splitting attribute for the node. After that, the same thread goes on to update the $node\_ids$ array. In the end, each thread is again dynamically assigned to one or more attribute lists and splits the attribute list. There are two synchronizations in one such process (i.e. from the start of evaluation to the end of splitting): one after the evaluation and another one after the splitting.

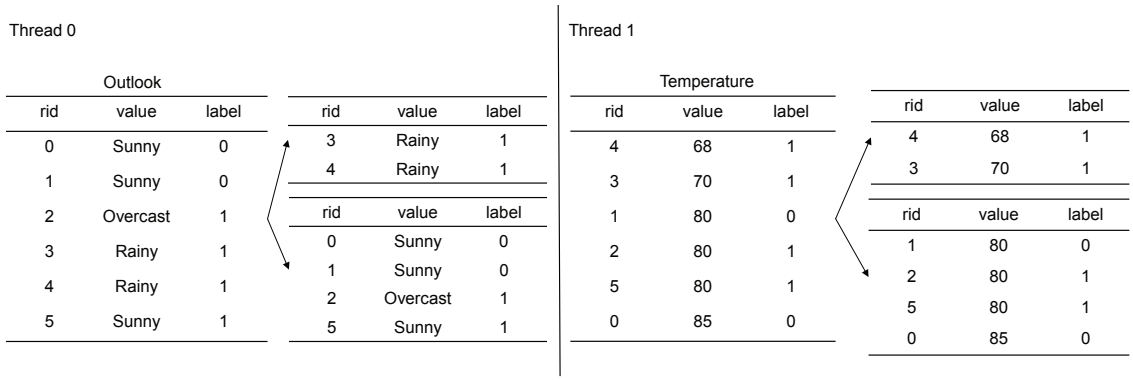


Figure 3.5: Attribute-level parallelism, each thread processes a whole attribute list.

3.2.2 Record-level Parallelization

In contrast to attribute-level parallelism, record-level parallelism (Figure 3.6) splits the attribute lists horizontally, and each thread gets a chunk (several rows) of all the attribute lists. We distribute the data in such a way that, if there are N records and m threads, each of the first $m-1$ threads gets $N/(m-1)$ records, and the last thread gets all the rest of the records. Since different threads evaluate different chunks of data that have different initial values for the count matrices. A parallel prefix sum is used to calculate the initial values of count matrices for different threads. After that, each thread evaluates its own chunks of data and maintains its own best splitting attribute. This requires two scans of the data (one for prefix sum, one for evaluation), but allows full parallelization. Later a thread compares the information gains and determines the global best splitting attribute as well as updating the $node\_ids$. There are three synchronizations in one process: one after the initialization of count matrices, one after the evaluation and one after the splitting.

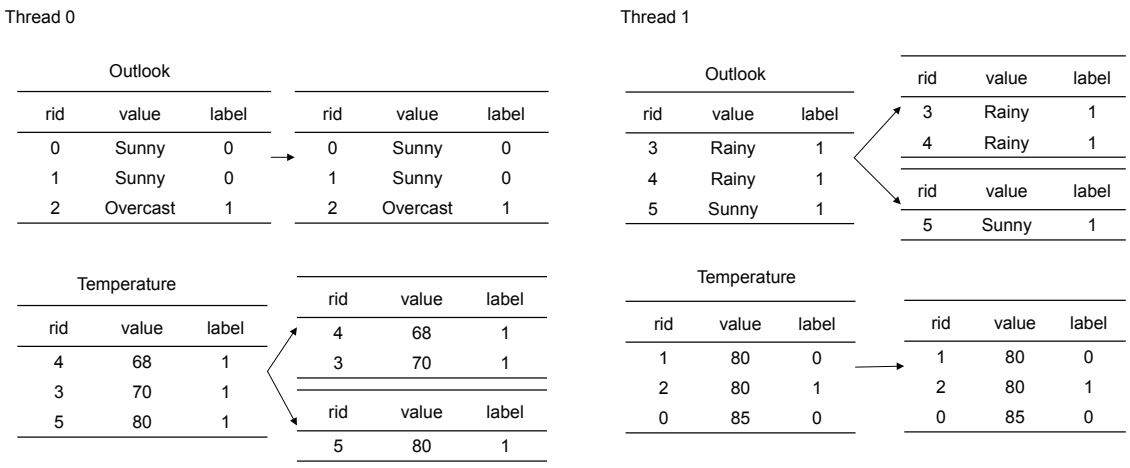


Figure 3.6: Record-level parallelism, each thread processes a chunk of all attribute lists.

3.2.3 Depth-first Growth

Depth-first growing refers to the case where a tree grows node-by-node in a depth-first and recursive way, i.e. all child nodes of a node are processed before processing another node. For example, the left tree in Figure 3.7 would grow in the following order: $0 \rightarrow 1 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3$. This results in two (if applying attribute-level parallelism) or three (if applying record-level parallelism) synchronizations per node.

3.2.4 Breadth-first Growth

We use breadth-first growth to refer to the situation where a tree grows level-by-level. That is, rather than growing one sub-tree after another, it grows a whole level of the tree at a time. The growing order of the nodes of the right tree in Figure 3.7 would be $0- > 1, 2, 3- > 4, 5$. This approach synchronizes two or three times (depends on the parallelism type) *per level*. This is better than the depth-first growth when the tree is bushy.

3.3 Parallel Algorithms

To parallelize the Algorithm 3.0, we combined the above parallelism types and tree growing patterns, which gives us four different parallel algorithms (algorithm 3.1 to 3.4). Their characteristics are shown in Table 3.1.

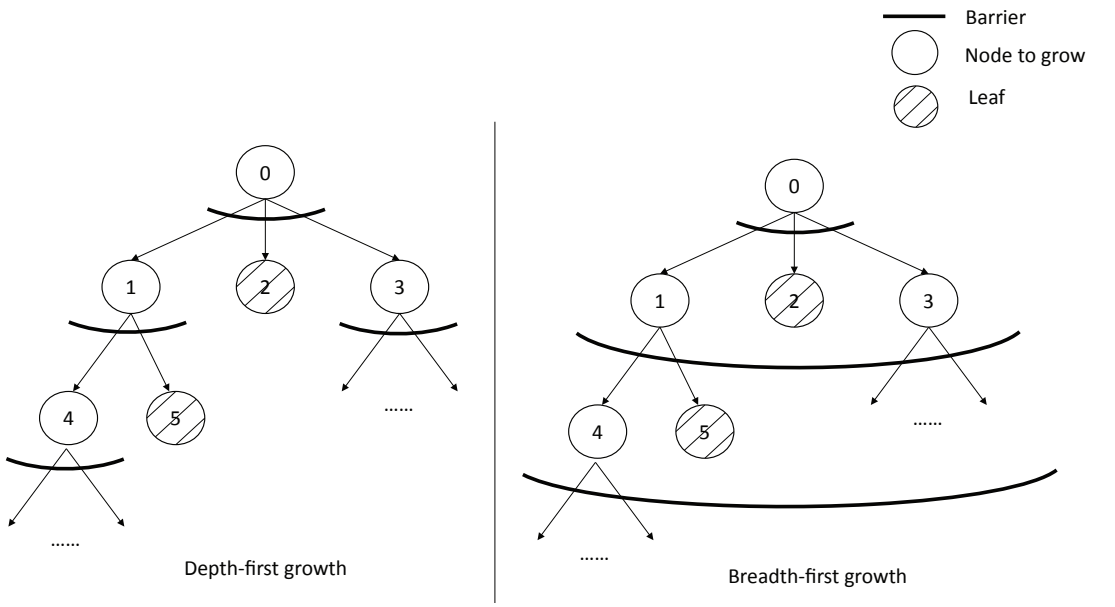


Figure 3.7: Tree growing patterns. Depth-first growth synchronize per node, breadth-first growth synchronize per level.

Table 3.1: Parallel algorithms

Algorithm	Parallelism type	Growing pattern
parallel_AD	attribute-level	depth-first
parallel_AB	attribute-level	breadth-first
parallel_RD	record-level	depth-first
parallel_RB	record-level	breadth-first

Assume that we have $n\_threads$ threads in total. For parallel_AD (Algorithm 3.1), each thread will have what it considers to be the best splitting attribute and the information gain of it after the evaluation. Then one thread sequentially finds the best splitting attribute from the $n\_threads$ best information gains coming from all threads. For parallel_AB (Algorithm 3.2), similarly, each thread will have what it considers to be the best splitting attribute for each node on the current level, and a thread finds the best splitting attribute for each node on the level. For both of the two algorithms, each thread is responsible for splitting one or several whole attribute list(s) so we keep one table of records per attribute per node.

For record-level algorithms, however, things are different. For parallel_RD (Algorithm 3.3), each thread is responsible for a part of each attribute list. When splitting the attribute lists, each thread will work on their own chunk of data and split it into two tables. Thus, we keep one table per

attribute per thread per node. It makes the data distribution and evaluation in the next call to growTree more complicated since threads may need to access to tables that come from a splitting performed by other threads in the previous splitting of the parent node. The same thing applies to parallel_RB (Algorithm 3.4).

Algorithm 3.1 parallel_AD

```
growTree (TreeNode n):
    if (n is a leaf):
        return
    end if

    for each attribute a ∈ A do in parallel:
        Evaluate a
    end for
    synchronize

    Find the best splitting attribute
    Update nodes_ids

    for each attribute a ∈ A do in parallel:
        Split a by nodes_ids
    end for
    synchronize

    for each child node c of n:
        growTree(c)
    end for
```

Algorithm 3.2 parallel_AB

```
growTree(TreeNodeQueue Q):
    for each node n in Q do:
        if (n is a leaf):
            return
        end if
    end for

    for each attribute a ∈ A do in parallel:
        for each node n in Q:
            Evaluate a
    end for
    synchronize

    for each node n in Q do:
        Find the best splitting attribute
        Update nodes_ids
    end for

    for each attribute a ∈ A do in parallel:
        for each node n in Q_c do:
            Split a by nodes_ids
        end for
    end for
    synchronize

    for each node n in Q do:
        for each child node c of n do:
            Q_c.push(c)
        end for
    end for
    growTree(Q_c)
```

Algorithm 3.3 parallel_RD

```
growTree(TreeNode n):
    if (n is a leaf):
        return
    end if

    for each data chunk d do in parallel:
      for each attribute a ∈ A do:
        Count class frequencies in d_a
      end for
    end for
    synchronize
    Calculate initial count matrices

    for each data chunk d do in parallel:
      for each attribute a ∈ A do:
        Evaluate d_a
      end for
    end for
    synchronize

    Find the best splitting attribute
    Update nodes_ids

    for each data chunk do in parallel:
      for each attribute a ∈ A do:
        Split d_a by nodes_ids
      end for
    end for
    synchronize

    for each child node c of n do:
      growTree(c)
    end for
```

Algorithm 3.4 parallel_RB

```
growTree(TreeNodeQueue Q):
    for each node n in Q do:
      if (n is a leaf):
          return
      end if
    end for

    for each data chunk d do in parallel:
      for each attribute a ∈ A do:
        Count class frequencies in d_a
      end for
    end for
    synchronize
    Calculate initial count matrices

    Each thread t do in parallel:
        for each node n in Q do:
            for each attribute a ∈ A do:
                evaluate chunk a_t
            end for
        end for
    synchronize

    for each node n in Q do:
        Find the best splitting attribute
        Update nodes_ids
    end for

    Each thread t do in parallel:
        for each node n in Q do:
            for each attribute a ∈ A do:
                Split a by nodes_ids
            end for
        end for
    synchronize

    for each node n in Q do:
        for each child node c of n do:
            Q_c.push(c)
        end for
    end for
    growTree(Q_c)
```

3.4 Evaluation

This section evaluates the performance of the sort-based algorithms.

3.4.1 Experimental Set-up and Test Data

The experiments of the sequential algorithm are run on a machine with an Intel i5 CPU 750 @ 2.67GHz and 4GB RAM. It has Ubuntu 12.04 with Linux kernel version 3.2.0-23-generic as the operating system. For this evaluation, eight medium-sized datasets from the UCI repository[1] are used. The characteristics of the datasets are summarized in Table 3.2. Ten-fold cross validation was applied to each of the datasets and each experiment was repeated 5 times. Results reported below are all average values, and the variance is very small so we do not show it in the plot.

Table 3.2: Test data for sequential algorithms

Dataset	Number of instances	Number of numerical attributes	Number of categorical attributes	Total number of attributes	Number of classes
abalone	4177	7	1	8	29
waveform-+noise	5000	40	0	40	3
nursery	12960	0	8	8	5
magic04	19020	10	0	10	2
shuttle	43500	9	0	9	7
connect-4	67557	0	42	42	3
Skin_NonSkin	245057	3	0	3	2
covtype	581012	10	44	54	7

For the implementation of parallel algorithms, we have access to a server that has an Intel CPU Xeon E5620 @ 2.40 GHz and 264GB of RAM. We used one socket and up to 8 physical cores with hyperthreading disabled. Each core has a 256KB L2 cache and all eight cores share a 20MB L3 cache. It has Debian 8 with Linux kernel version 3.18.14-U as the operating system. The medium large dataset that was used for testing parallel algorithms is shown in Table 3.3.

[1] http://archive.ics.uci.edu/ml/

Table 3.3: Test data for parallel algorithms

Dataset	Number of instances	Number of numerical attributes	Number of categorical attributes	Total number of attributes	Number of classes
KDD Cup	4000000	32	9	41	23

3.4.2 Sequential Algorithm

In order to check the quality of our sequential algorithm (Algorithm 3.0), we compare it with Christian Borgelt's decision tree [5] implementation and the YaDT algorithm [30], both of which are publicly accessible.

To make our description easier, we divide the whole execution process into three parts: data loading, pre-sorting, and tree growing. Figure 3.8 and 3.9 show the relative error rate and relative total execution time (including data loading, sorting and tree growing) of different sequential algorithms. Here we refer to our sequential algorithm as sequential. The results show that our sequential algorithm is comparable with other sequential algorithms in terms of both accuracy and efficiency. This makes our sequential algorithm a sound basis for building a parallel one.

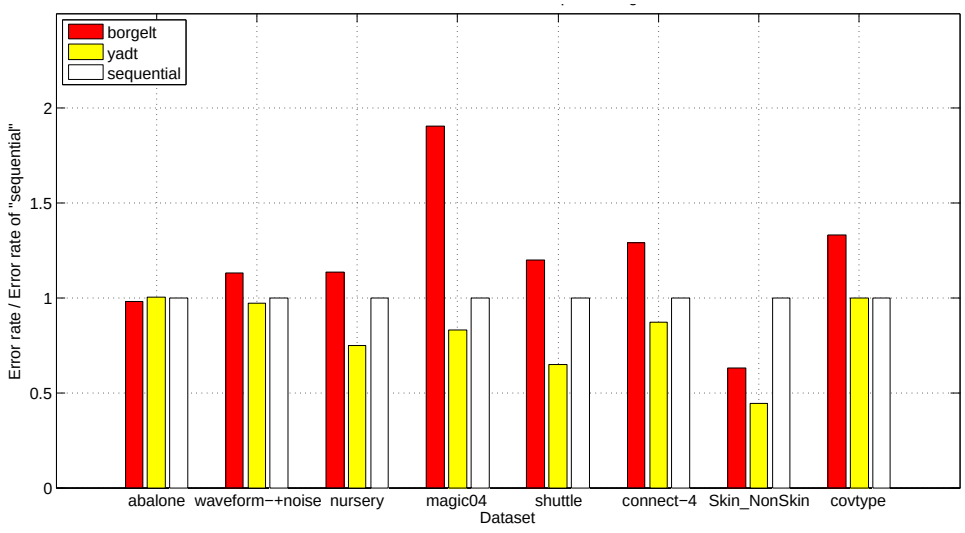


Figure 3.8: Relative error rate of sequential algorithms.

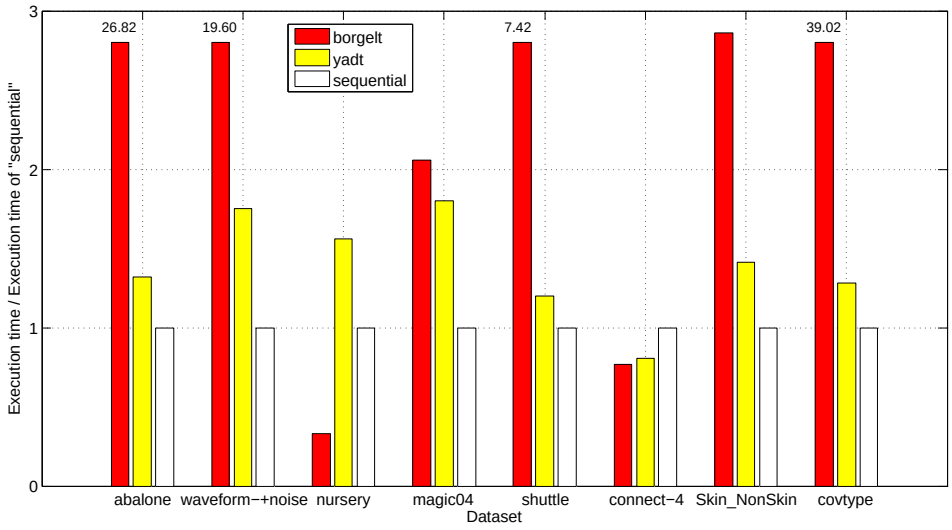


Figure 3.9: Relative total execution time of sequential algorithms. This includes data loading, pre-sorting and tree growing time.

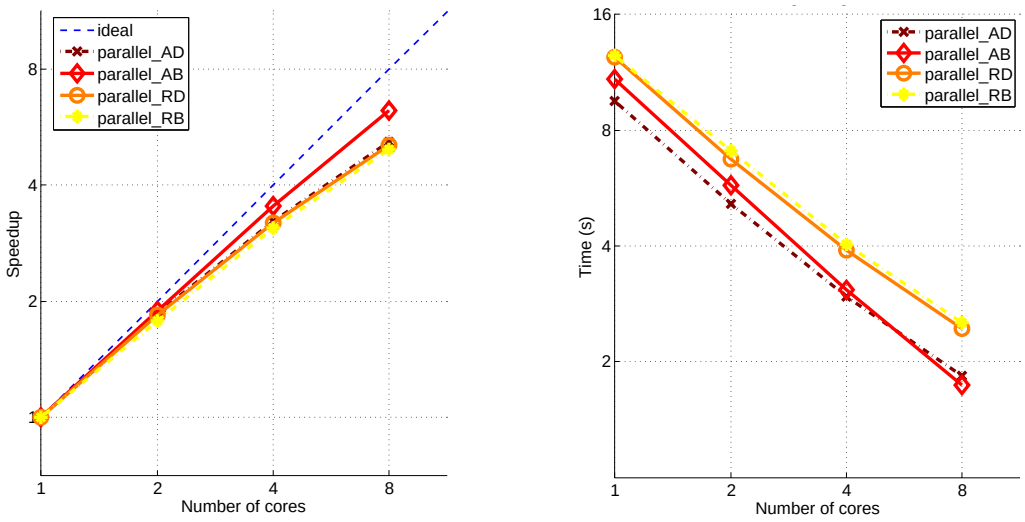
3.4.3 Parallel Algorithms

For parallel algorithms, we focus our analysis on the tree growing part. Although pre-sorting also takes an amount of time (we discuss where time is spent in more details below, Figure 3.12) that is not negligible, there are existing parallel sorting algorithms that may be applicable to optimize this, and it is not our premier intention to target at this. The speedup of tree growing time of different parallel algorithms (with respect to each single-threaded version) are shown in Figure 3.10 (a). The best result comes from parallel_AB which achieves x6.3 speedup when using eight threads. While there are some common problems for all algorithms, we will first discuss the specific problems that each algorithm has and come back to the common problems later.

We start the analysis by breaking the tree growing part into four phases:

- Evaluation - evaluation of attributes
- Updating - updating the position array $node\_ids$
- Splitting -splitting the attribute lists into child lists
- Others - time consumed between calls to the growTree function. This includes: pushing parameters and return addresses onto stack, branching to destination functions, creating new stack frames, undoing the above at the end of the function and destroying local variables, etc

Time of different phases are measured by adding timestamps between functions. Time of the *Others* phase is calculated by subtracting the execution time of the other three phases from the total



(a) (b)

Figure 3.10: Speedup and tree growing time of sort-based algorithms.

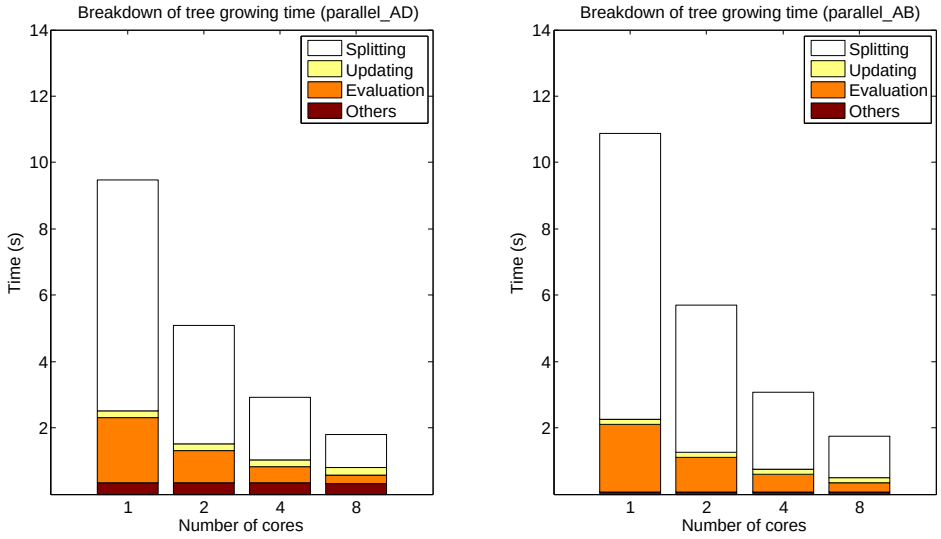
growing time.

Depth-first vs. Breadth-first

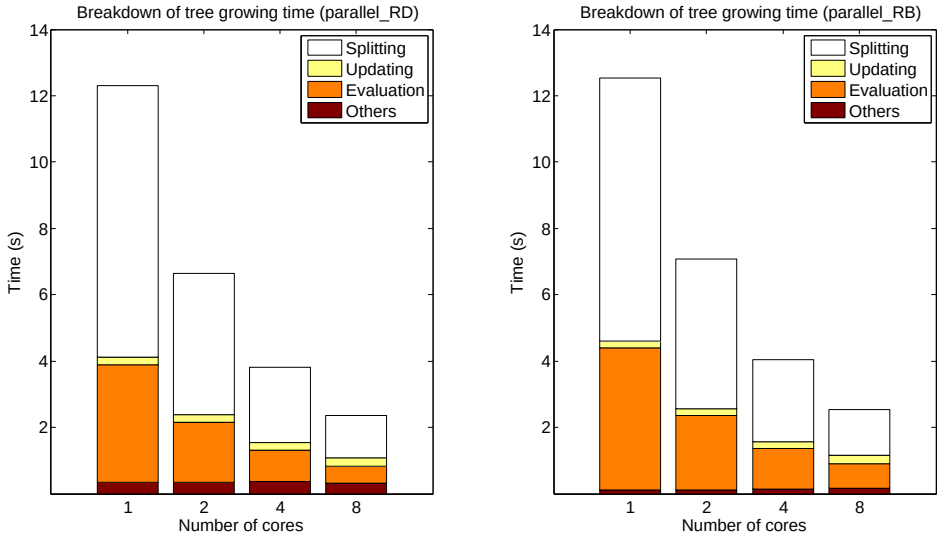
As shown in Figure 3.11, depth-first algorithms have a more prominent *Others* part than the breadth-first algorithms. This is because the depth-first algorithms call the growTree function once per node, whereas the breadth-first algorithms only call it once per level. This means depth-first algorithms need to do all the work in *Others* more times than breadth-first algorithms. Since the number of nodes in a tree is always larger than the depth of it, depth-first algorithms will always have this disadvantage over breadth-first algorithms. However, by comparing the single-threaded versions of parallel_AD and parallel_AB we can see that parallel_AD runs slightly faster than the single-threaded parallel_AB. This is because that we lose some locality when growing the tree in breadth-first order. When growing in depth-first order, we can use the data at a node for growing its subtrees if the node is small enough for the data to fit in cache. However, in breadth-first algorithms, we do not grow the tree recursively and thus lose this locality.

Attribute-level vs. Record-level

Figure 3.10 (b) shows the tree growing time of different parallel algorithms. We notice that the record-level parallelized algorithms take about 20% longer than their attribute-level parallelized counterparts. This is expected since horizontal algorithms are more complicated in terms of program structure, distributing data and finding the best splitting attribute (as discussed in Section 3.3). More importantly, we need to calculate the initial value of the count matrices for each differ-



(a)



(b)

Figure 3.11: Breakdown of tree growing time of sort-based parallel algorithms.

ent threads, so that later each thread can evaluate on its own chunk of data independently. Even though we parallelized it by having each thread count its own chunk of data and adding the results one by one when all threads are done, it still requires an extra scan over all the numerical attribute

lists compared to the attribute-level parallelized algorithms. Apart from that, if the distribution of the candidate splitting points of numerical attributes is heavily skewed, there would also be the case where a few threads need to evaluate a majority of the candidate splitting points while other threads have very few points to evaluate, which could cause load imbalance in the *Evaluation* phase. This problem might be solvable but since evaluation takes a modest amount of time, we did not address this issue.

Attribute-level parallelized algorithms, on the other hand, does not have the above problems. However, one potential drawback of it is that if the number of threads is larger than the number of attributes, some threads will go idle and do nothing, which makes it an inevitable bottleneck if we want to scale the algorithms on large-scale architectures.

Common problems

Figure 3.12 shows the breakdown of pre-sorting and tree-growing time of the sequential algorithm and different parallel algorithms run by one thread. We can see that the Splitting phase takes the majority of time in all algorithms compared to any other phases. The reason behind this is that in the Splitting phase, all algorithms need to access the $node\_ids$ array that stores the child node ID for each record in the current table(s) by record IDs. For numerical attributes, this access is random because each numerical attribute is sorted differently and thus will access $node\_ids$ in different orders. This could let the program suffer from poor spatial locality in cache.

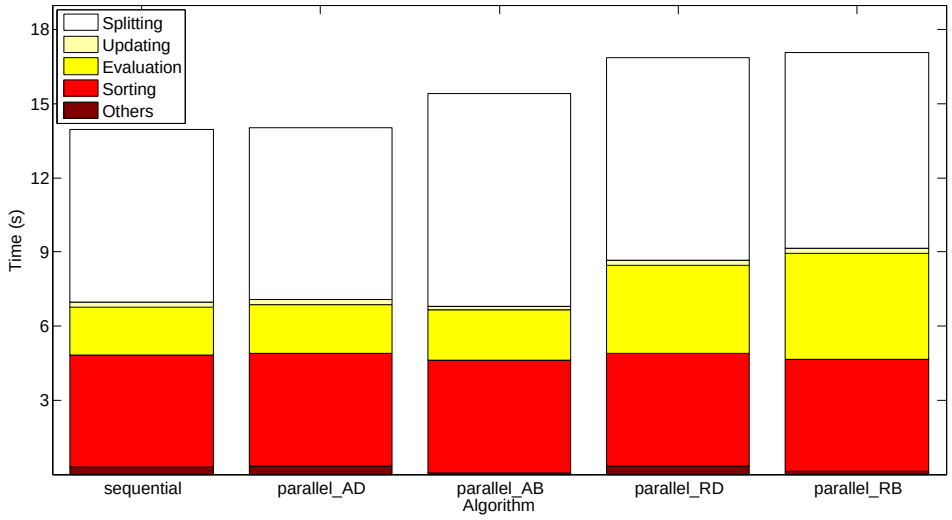


Figure 3.12: Breakdown of global time of sort-based algorithms run by 1 thread.

In order to show this locality problem, we run a micro benchmark. Figure 3.13 and Algorithm 3.5 illustrate how the micro benchmark works. We use $lists$ to simulate the attribute lists. Each

element in the $list$ is a tuple with three integers: rid, v and c. The first integer, rid, is a random number within the range of 0 to $n - 1$, where n is the size of $node\_ids$. The other two integers, v and c, are used to simulate attribute value and class label respectively. They are initialized to zeroes.

Recall that the *Splitting* phase has three steps: reading a record from an attribute list, fetching the child node id from $node\_ids$ according to the records rid, and writing the record to the child table. To make the analysis simpler, we simulate the first two reading steps of the *Splitting* phase. The algorithm iterates over all $lists$. In each iteration, it scans all the elements in a $list$ and accesses $node\_ids$ by the elements rid. There are 40 $lists$ in total and the experiment is run 5 times by one thread.

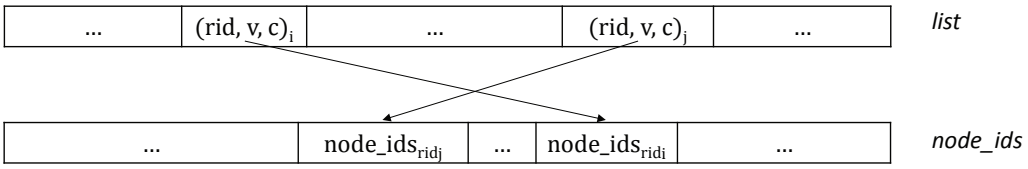


Figure 3.13: Micro benchmark for accessing $node\_ids$, the $node\_ids$ array is accessed according to the random rids in $list$, the same way as our parallel decision tree algorithms do.

Algorithm 3.5 Accessing *node_ids*

```
Initialize lists, node_ids
for each list in lists do:
    for each element e in list do:
        nid = node_ids[e.rid]
    end for
end for
```

Figure 3.14 shows the L2 and L3 cache misses per element accessed as well as the access time as functions of the size of $node\_ids$. It is clear to see that there is a leap in L2 cache misses when $node\_ids$ has size about 200KB, which is around the size of L2 cache. So when it grows larger and does not fit in L2 cache, random access to it results in a large number of cache misses, which also causes the access time to become longer. A similar scenario for L3 cache misses that occurs around the point where the size of $node\_ids$ is close to the size of L3 cache is due to the same reason, and causes an even more significant jump in the access time. Our large dataset has about 4.4M records, which means that the $node\_ids$ has size of 17.6MB and that is very close to the size of L3 cache. As a result, random access to it is expensive and causes the Splitting phase to be the most time-consuming part of the algorithms.

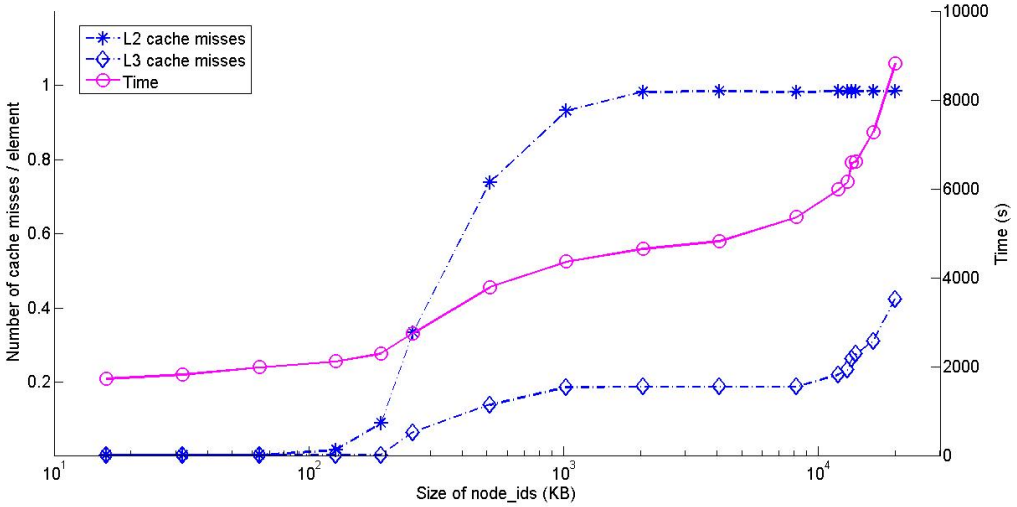


Figure 3.14: Performance of accessing $node\_ids$ (single thread).

Apart from consuming a lot of time, the *Splitting* phase also does not scale very well. We approach this problem by running the above micro benchmark using eight threads in parallel (see Algorithm 3.6), so that each thread gets several lists and accesses $node\_ids$ in independent random orders. This is exactly how the parallel decision tree algorithms access the $node\_ids$ in the *Splitting* phase. Figure 3.15 shows the speedup and cache misses per element at different sizes of $node\_ids$. We see that the speedup is very close to 8 until the point where the number of L3 cache miss starts to rise. After that it drops by about 10%. This indicates that the speedup of the *Splitting* phase is bounded by the performance of cache. As already mentioned before, the micro benchmark only simulates the reading process in Splitting. In the real decision tree algorithms, there is still a writing process (writing a data record to a child table) which would bring more data (i.e. the target child table to be written to) into cache and further degrade the performance.

Algorithm 3.6 Accessing *node_ids* (parallelized)

```
Initialize lists, node_ids
for each list in lists do (in parallel):
    for each element e in list do:
        nid = node_ids[e.rid]
    end for
end for
```

Another problem that is worthy of attention is that all of the algorithms have the sequential *Updat-*

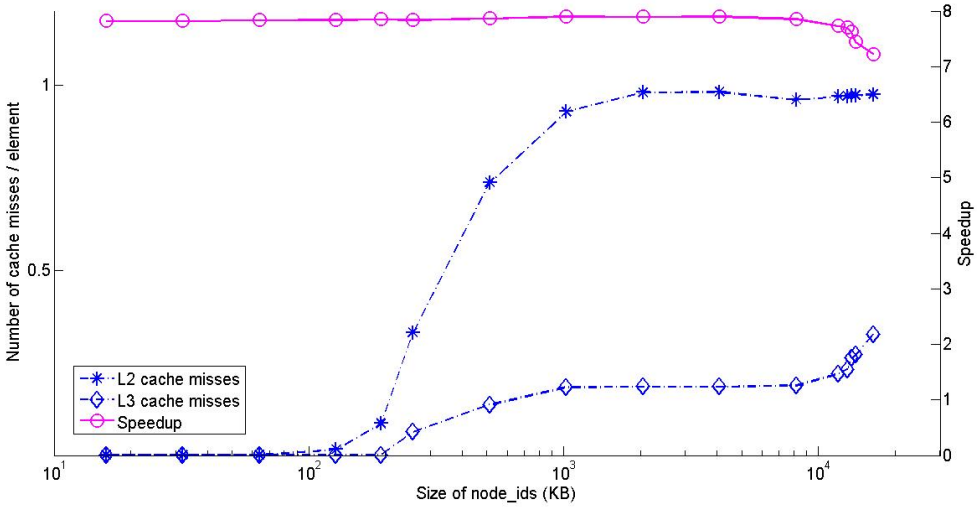


Figure 3.15: Performance of accessing $node\_ids$ (eight threads).

ing phase that does not scale and becomes more of a bottleneck as the number of threads grows (Figure 3.11). While it may be possible to parallelize it to some extent, because of the poor performance of the *Splitting* phase we did not take this further.

In summary, the random access to $node\_ids$ results in increasing cache misses as the size of $node\_ids$ increases and deteriorates both the absolute running time as well as the speedup. It is necessary for sort-based algorithms since numerical attributes are pre-sorted and we need to keep them in order throughout the whole tree growing process. However, this is too expensive and limits the scalability of the algorithm. Thus, we investigate algorithms that do not require pre-sorting in the next chapter.

Chapter 4

Count-based Decision Tree Induction

This chapter presents the count-based decision tree induction algorithms. Section 4.1 introduces AVC-group and the naïve count-based algorithms. Section 4.2 describes the idea of partitioning and its application in our algorithms. Section 4.3 evaluates the performance of different count-based algorithms and compares them to the sort-based ones.

4.1 Count-based Algorithm

This section gives a fundamental introduction of the count-based algorithms.

4.1.1 AVC-group

The RainForest algorithm by [11] first proposed the method to use the frequency of (Attribute value, Class label) pairs to avoid pre-sorting of the numerical attributes, which makes the splitting phase very expensive. The AVC-set of a numerical attribute at a node is a table that summarizes all the distinct (Attribute value, Class label) pairs that appear in the original data and the frequency of each pair. Once we have the AVC-set and sort it by attribute value, we can go through the sorted AVC-set and evaluate each candidate splitting point as before. The AVC-set is a compressed representation of the original data that provides enough information needed for calculating the information gains at each candidate splitting points (see Figure 4.1 for an example). The AVC-group is defined as the set of all AVC-sets at a node.

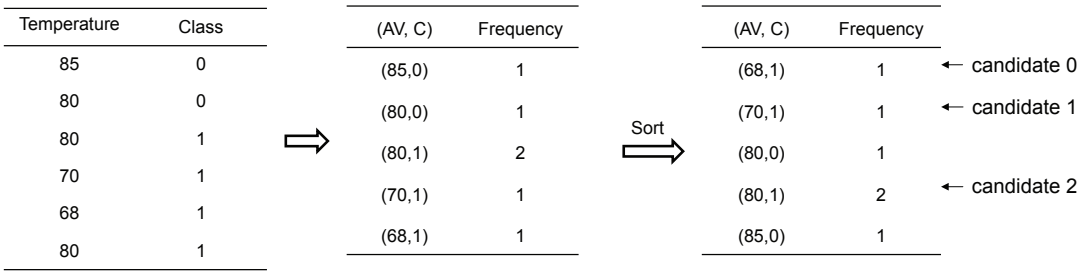


Figure 4.1: AVC-set for attribute *Temperature* at the root node and the use of it for evaluation

4.1.2 Sequential Algorithm

Since the count-based algorithm does not need to sort the original data, the representation of the data becomes simpler than the sort-based algorithm. We no longer need to keep the $record\_id$ explicitly and the data is represented as the original relational table with each column as an attribute and each row as a record (Figure 3.1). The table is stored column-wise. This feature of not having to stay sorted also allows the access to the $node\_ids$ array in the splitting phase to be sequential.

Apart from that, the only difference between the count-based sequential algorithm and the sort-based sequential algorithm is the handling of numerical attributes. Unlike the sort-based algorithm, which first sorts the numerical attributes directly, scans the sorted attribute lists and evaluates each candidate splitting point, the count-based algorithm scans the unsorted data and constructs an unsorted AVC-set for the attribute. Later it sorts the AVC-set, scans it and evaluates each candidate splitting point. To evaluate one numerical attribute, the count-based algorithm requires one scan over the original data and one scan over the sorted AVC-set. Assuming that there are n records in the data and m elements in an AVC-set, the complexity of sorting the AVC-set and evaluating is then $O(mlog(m)) + O(n)$. In contrast, the sorting and evaluation of an attribute in the sort-based algorithm has a complexity of $O(nlog(n)) + O(n)$. In reality, m is very often less than n (see Table 4.1). Thus, the count-based algorithm typically has lower complexity than the sort-based algorithm.

4.1.3 Parallel Algorithm

We use the record-level parallelization scheme used in parallel_RD to parallelize the count-based algorithm (Algorithm 4.0). For each attribute, each thread scans a chunk of data and constructs a sorted AVC-set (if the attribute is numerical) or a count matrix (if the attribute is categorical). For the sake of concision, we use the term *evaluation structure* to denote either an AVC-set or a count matrix, as they are both used for the evaluation of an attribute. After that, we use a parallel reduction tree to merge all the *evaluation structures*. There is a synchronization at each level of

Table 4.1: Distribution of the size of AVC-sets for some datasets

Number of elements in an AVC-set	Number of attributes			
	KDD Cup	covtype	magic04	shuttle
m	$n = 4M$	$n = 581K$	$n = 19K$	$n = 43K$
0˜10	1	0	0	0
10˜100	2	0	0	0
100˜1K	5	1	0	7
1K˜10K	19	6	1	2
10K˜100K	2	3	9	0
100K˜1M	3	0	0	0

the reduction tree, which leads to a total $log_2(n\_threads)$ synchronizations for the whole merging phase of one numerical attribute..

Now that we have the data structures that are needed for the evaluation, we can evaluate all the attributes and split the node. Each thread keeps a single table of the original records per attribute per node and the splitting is the same as the splitting in parallel_RD algorithm. Once the splitting is done, the growTree() method is called on each of the children.

It can be easily calculated that there are $(2+n\_numerical\_attributes \times log_2(n\_threads))$ synchronizations for processing each node ($n\_numerical$ is the number of numerical attributes).

Algorithm 4.0 parallel_AVC

growTree ($TreeNode\ n$):
  if (n is a leaf):
    return
  end if

  for each numerical attribute $a \in A$ **do**:
    for each data chunk d **do (in parallel)**:
      Construct and sort AVC-set avc_a^d
    end for
  end for

  for each categorical attribute $a \in A$ **do**:
    for each data chunk d **do (in parallel)**:
      Construct count matrix mat_a^d
    end for
  end for
  synchronize

  mergeAVCSetsAndMatrices($\left\{ \boldsymbol{avc}_{a_i}^{p}, \boldsymbol{mat}_{a_j}^{p} \;\middle|\; \boldsymbol{a_i}, \boldsymbol{a_j} \in \boldsymbol{A},\ \boldsymbol{t} = 0,1, \ldots, \boldsymbol{n\_threads} - 1 \right\}$)

  for each attribute $a \in A$ **do**:
    Evaluate a
  end for

  Find the best splitting attribute
  Construct position array $node\_ids$

  for each attribute $a \in A$ **do**:
    for each data chunk d_a **do (in parallel)**:
      Split d_a by $nodes\_ids$
    end for
  end for
  synchronize

  for each child node c **do**:
    growTree(c)
  end for

Algorithm 4.0.1 mergeAVCSetsAndMatrices (
$\left\{ \boldsymbol{avc}_{a_i}^{p}, \boldsymbol{mat}_{a_j}^{p} \middle| \boldsymbol{a_i}, \boldsymbol{a_j} \in \boldsymbol{A},\ \boldsymbol{t} = 0,1,\dots, \boldsymbol{n\_threads} - 1 \right\}$**)**

```
mergeAVCSetsAndMatrices ()
    for each numerical attribute a ∈ A do:
        step_size = 2
        while step_size <= n_threads do (in parallel):
            for i = 0, step_size, step_size + step_size, ...
                Merge
            end for
            synchronize
            step_size = step_size * 2
        end while
    end for
    for each categorical attribute a ∈ A do:
        step_size = 2
        while step_size <= n_threads do:
            for i = 0, step_size, step_size + step_size, ... do (in parallel):
                Merge
            end for
            synchronize
            step_size = step_size * 2
        end while
    end for
```

4.2 Count-based Algorithm Using Partitions

This section introduces partitioning and how it is applied to the count-based decision tree algorithms.

4.2.1 Partitioning

The challenge of the above count-based algorithm comes when the size of the AVC-group is large. Constructing and sorting large AVC-groups can be cache inefficient and limit parallel performance. In order to solve this problem, the SPIES algorithm [14] uses statistical sampling to obtain memory-efficient processing of numerical attributes. However, this approach requires another pass on the data if false pruning occurs. SPIES also only focuses on the growing of large nodes and stop growing a node if there is less than 1M records. To avoid the overhead of false pruning and focus on the growing of the whole tree, we propose an algorithm which combines the AVC-group idea and partitioning to achieve cache efficiency and requires only one scan of the original data.

The key of the new algorithm is the use of tuned partitioning routine (presented by [7, 21]) to distribute (Attribute value, Class label) pairs of numerical attributes to partitions. A partition is a set of (Attribute value, Class label) pairs. When scanning the data, a hash function is applied to each record‘s (Attribute value, Class label) pair and the hash value is the index of the partition to which the record is going. For simplicity reason, we use the following commonly used hash function for pairs, where hash() is the C++ default standard hash function that is implementation dependent:

$$partition\_id = (hash(Attributevalue) \oplus hash(Classlabel)) \mod (n\_partitions)$$

Figure 4.2 gives an illustration of the partitioned *Temperature* attribute. When constructing an AVC-set, instead of keeping one AVC-set per attribute, we construct one AVC-set per partition, thus reducing the cache footprint required by each AVC-set.

4.2.2 Sequential Algorithm

Algorithm 4.1 shows the sequential version of the new algorithm. The algorithm consists of three parts:

- Processing numerical attributes. The processing of each numerical attribute contains four parts: Partitioning: scanning the attribute values and distribute each record to a partition according to the hash value of the records (Attribute value, Class label) pair; Constructing AVC-sets: after partitioning, an AVC-set is constructed from each partition. An AVC-set is constructed by scanning all the (Attribute value, Class label) pairs in a partition and count the

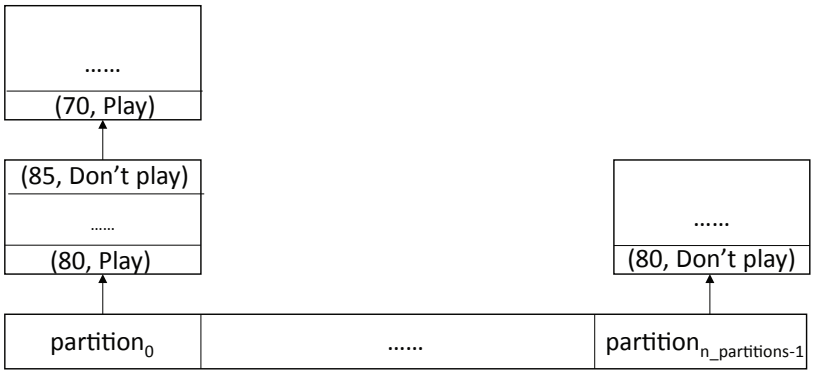


Figure 4.2: Illustration of a partitioning buffer for attribute *Temperature*. The partitioning buffer consists of an array of blocks (partitions) and a free pool. Each partition is a chain of fixed-size blocks and each entry inside the block is an (Attribute value, Class label) pair.

frequencies of each distinct (Attribute value, Class label) pair. The AVC-set is then sorted by attribute value; Merging AVC-sets: appending all sorted AVC-sets into one AVC-set; Evaluating: scanning the AVC-set and evaluate each candidate splitting point. The evaluation criteria is the same as the sort-based algorithm. Figure 4.3 shows an example of procedures 1) - 3).

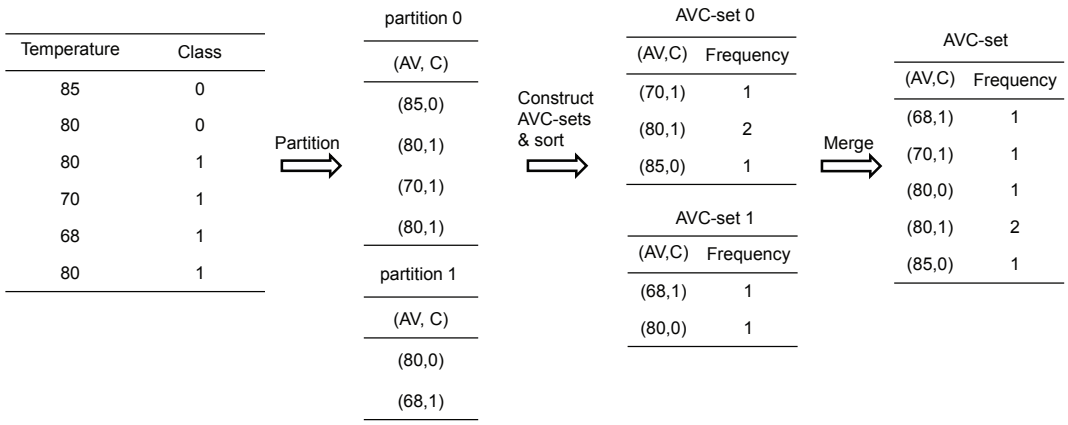


Figure 4.3: Illustration of the handling of numerical attribute using partitioning buffers

- Processing categorical attributes. The number of distinct values of a categorical attribute is typically small. Thus, there is no need to use partitioning. Categorical attributes are evaluated in the same way as the sort-based algorithm. That is, for each attribute, scanning the attribute values whilst constructing the count matrix and evaluating the count matrix after the scan.

- Splitting. After getting the best splitting attribute, we scan the splitting attribute and construct the index array $node\_ids$. Then the table at the current node is split into sub-tables according to this array.

Algorithm 4.1 sequential_AVC_P

growTree ($TreeNode\ n$):
  if (n is a leaf):
    return
  end if

  for each numerical attribute $a \in A$ **do**:
    Partition a into partitions of (AV, C) pairs: $partition_0, \ldots, partition_{n\_partitions\ -1}$
    Construct AVC-sets out of partitions: $avc_a^0, avc_a^2, \ldots, avc_a^{n\_partitions\ -1}$
    Merge AVC-sets to avc_a
    Evaluate avc_a
  end for

  for each categorical attribute $a \in A$ **do**:
    Construct count matrix mat_a
    Evaluate mat_a
  end for

  Find the best splitting attribute
  Construct position array $node\_ids$

  for each attribute $a \in A$ **do**:
    Split a by $node\_ids$
  end for

  for each child node c **do**:
    growTree(c)
  end for

4.2.3 Parallel Algorithm

Data parallelism

Our first attempt to parallelize the sequential_AVC_P algorithm leads to a record-level data parallelism scheme.

The algorithm begins by partitioning each numerical attribute. The idea of record-level parallelism

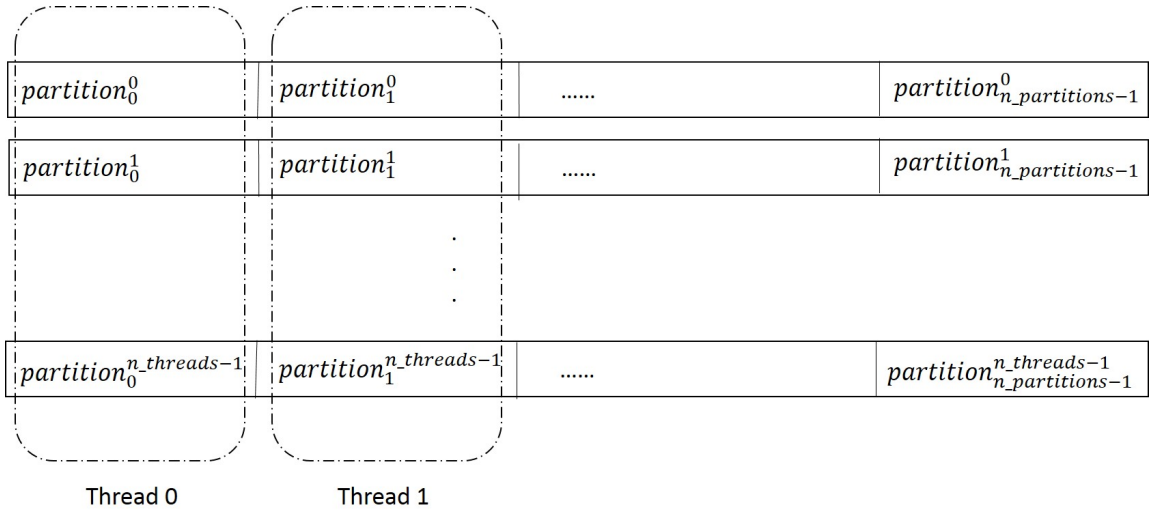


Figure 4.4: Partitioning result of a numerical attribute. Each row represents a set of partitions of a thread, all partitions in the same column have the same partition ID. When constructing AVC-sets, each thread is responsible for all the partitions with the same partition id, i.e. a column (or several columns) in the grid (indicated by the dotted lines).

introduced in Section 3.2 is used here. The partitionNumericalAttribute() method loops over all numerical attributes and for each numerical attribute, each thread handles a chunk of the attribute data and does the partitioning on its own. This produces $n\_threads$ sets of partitions with $n\_partitions$ partitions each (Figure 4.4).

After partitioning, the constructAVCSetsAndMatrices() method constructs the evaluation structures. For a numerical attribute, each thread dynamically picks one set of partitions that have the same partition ID (Figure 4.4), scans the (Attribute value, Class label) pairs inside the partitions and constructs an unsorted AVC-set. After the scan is finished, it sorts the AVC-set. This produces $n\_partitions$ AVC-sets in total for one numerical attribute.

The construction of the count matrices for the categorical attributes is very similar. For a single categorical attribute, each thread creates a count matrix for a chunk of data, which leads to $n\_threads$ count matrices in total.

Algorithm 4.2 shows the parallel_AVC_PD algorithm, there are $(3+n\_numerical \times log_2(n\_partitions))$ synchronizations per node.

Algorithm 4.2 parallel_AVC_PD

growTree ($TreeNode\ n$):
    if (n is a leaf):
        return
    end if

    partitionNumericalAttributes ()
    synchronize

    $\{avc_a^p, mat_a^p \mid a \in A,\ p = 0,1,\dots,n\_partitions - 1\}$=**constructAVCSetsAndMatrices** ()
    synchronize

    mergeAVCSetsAndMatrices ($\{avc_{a_i}^p, mat_{a_j}^p \mid a_i, a_j \in A,\ p = 0,1,\dots,n\_partitions - 1\}$)

    for each attribute $a \in A$ **do**:
        Evaluate a
    end for

    Find the best splitting attribute
    Construct position array $node\_ids$

    for each attribute $a \in A$ **do**:
        Split a by $node\_ids$
    end for
    synchronize

    for each child node c **do**:
        growTree(c)
    end for

Algorithm 4.2.1 partitionNumericalAttributes ()

partitionNumericalAttributes ()
    for each numerical attribute $a \in A$ **do**:
        for each data chunk d **do (in parallel)**:
            Get thread id t
            Partition d into partitions of (Attribute value, Class label) pairs:
                $partition_0^t, \dots, partition_{n\_partitions\ -1}^t$
        end for
    end for

Algorithm 4.2.2 constructAVCSetsAndMatrices ()

constructAVCSetsAndMatrices ()
    for each numerical attribute $a \in A$ **do**:
        for each set of partitions with the same partition id p
        (i.e. $partition_p^0, \dots, partition_p^{n\_threads\ -1}$) **do (in parallel)**:
            Construct AVC-set and sort: avc_a^p
        end for
    end for
    for each categorical attribute $a \in A$ **do**:
        for each data chunk d **do (in parallel)**:
            Get thread id t
            Construct count matrix mat^t
        end for
    end for

    return $\left\{ \boldsymbol{avc}_{\boldsymbol{a_i}}^{\boldsymbol{p}}, \boldsymbol{mat}_{\boldsymbol{a_j}}^{\boldsymbol{p}} \,\middle|\, \boldsymbol{a_i}, \boldsymbol{a_j} \in \boldsymbol{A},\ \boldsymbol{p} = 0,1, \dots, \boldsymbol{n\_partitions} - 1 \right\}$

Task parallelism

There are potentially three types of parallelism that could be employed to the building of a tree: attribute-level parallelism, record-level parallelism and node-level parallelism. However, the structure of the tree is irregular and we need to exploit any subset of the three types of parallelism. Static scheduling is not flexible enough to achieve this goal, so we propose another parallel algorithm which is based on the parallel_AVC_PD algorithm but combines the record-level data parallelism with the task parallelism scheme provided by Intel TBB [8].

The idea of task parallelism is to express the whole algorithm as different tasks of about the same size and to execute them as concurrently as possible. The scheduler employs a technique of work stealing. Each thread maintains a deque (double-ended queue) of available tasks. When a task is spawned by a thread, it is added to the threads task deque. When a worker thread is available, it may run any spawned task, including unrelated tasks created by other threads. The details of the scheduling algorithm could be found on[1].

Before going through the algorithm, we first introduce the terms that are used later in our description:

- block: a chunk of data, e.g. a subset of all the records of attribute temperature. This is the working unit of record-level parallelism.
- $block\_size$: the pre-defined size of a block. It indicates the minimum amount of work required for the record-level parallelism to be employed, and stays the same throughout the algorithm. If the number of records at a node is smaller than $block\_size$, there will be no record-level parallelism. If the number of records at a node is a multiple of $block\_size$, the data is divided into equal-sized blocks. If it is larger than $block\_size$ but is not a multiple of it, the data is divided into blocks of $block\_size$ and a block that has less than $block\_size$ records. We use n_blocks to denote the number of blocks at a node, which is determined by the formula: $n\_blocks = \lceil n\_records/block\_size \rceil$
- spawn: the action of spawning a created task. Threads do not have to wait for the spawned task to be finished to execute the following code unless wait_for_all is called.
- wait_for_all: wait for the previously spawned sub-tasks of the current task to be finished before continuing with the current task. Note that this is different from a barrier, because while waiting for a task to complete, the threads may work on other available tasks that are independent of the task that is waiting.

The algorithm (Algorithm 4.3) expresses the growing of the tree as several tasks of a minimum size. On the highest level, the growing of a tree is expressed as a set of ProcessNodeTasks. Each ProcessNodeTask is further divided into three major tasks:

- ProcessNumericalAttributeTask: create the AVC-set for a numerical attribute and evaluate

[1] https://software.intel.com/en-us/node/506295

that attribute. This includes: partitioning, constructing AVC-sets, merging AVC-sets and evaluating.

- ProcessCategoricalTask: create count matrix for a categorical attribute and evaluate that attribute. This includes: constructing count matrices, merging count matrices and evaluating.
- SplitAttributeTask: split an attribute

The dependencies of the major tasks are shown in Figure 4.5-4.6. The processing and splitting of

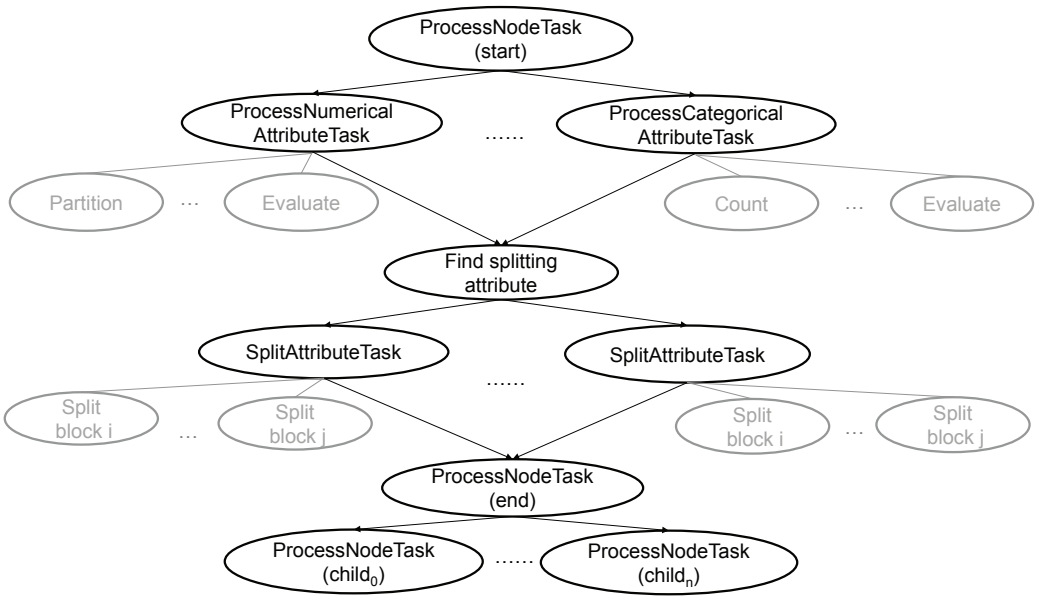


Figure 4.5: Task dependencies in ProcessNodeTask. A task may start only when all of its parents (along directed edges) are finished. Gray nodes represent tasks further spawned by the current tasks.

attributes in parallel_AVC_PT algorithm is very similar to parallel_AVC_PD. It processes and splits attributes in the same way parallel_AVC_PD does except for the following aspects:

1) The $block\_size$ is a fixed value in parallel_AVC_PT. This allows the algorithm to dynamically determine which levels of parallelism to apply when processing a node. If the number of records is large, both record-level and attribute-level parallelism are applied. If the number of records is less than the $block\_size$, all records will be handled by one thread and only attribute-level parallelism will be applied.

2) parallel_AVC_PT also explores the use of node-level parallelism. By spawning the tasks for processing child nodes at the end of each ProcessNodeTask, we have multiple nodes grown at the same time.

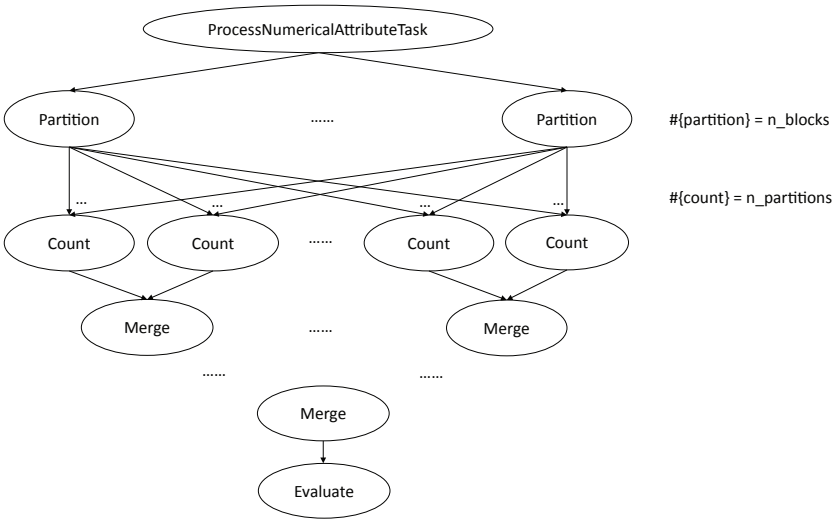


Figure 4.6: Task dependencies in ProcessNumericalAttributeTask.

3) There is no global barriers in parallel_AVC_PT. This reduces the waiting time of threads.

4) In terms of the representation of the original data, parallel_AVC_PD keeps one single array per thread per attribute per node while in parallel_AVC_PT, we simplify this by keeping only one table per attribute per node. Thus, in the splitting phase, each thread needs to know to which positions it should write the records. For this purpose, we construct a 2d array, $write\_idx$, before splitting in order to keep track of the correct writing positions in the sub-tables for each thread. The construction of $write\_idx$ is a parallel prefix sum. An illustration of it is given in Figure 4.7 Each thread scans a block of data and keeps $n\_children$ arrays of indices, where $n\_children$ is the number of children for the current node. For the tth block, the ith element in $write\_idx_c$ indicates the position to which the ith record in this block should be written if the record is going to child c. Note that the tth block needs to know the information in all the previous $t-1$ blocks in order to calculate the exact writing positions (Figure 4.7 (c)). However, this does not have to be done explicitly. We let each thread start from position zero and count the number of nodes that go to each child. Later, these numbers are prefix-summed to get the position offsets for each block. In the splitting phase, these offsets are summed up with the $write\_idx$ (like the one in 4.7 (b)) to finally determine the actual writing positions.

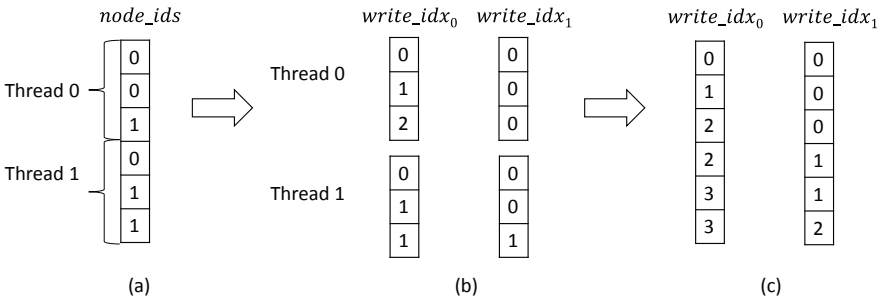


Figure 4.7: Use prefix-sum to construct writing position indices $write\_idx$, assuming that there are two child nodes. (a) Each thread takes a data block. (b) Thread scans the data block and for each record, it increments the $write\_idx$ for the child to whom the record is going to (starts from position zero). (c) Prefix-sum $write\_idx$ arrays from different threads to get the actual writing positions.

Algorithm 4.3 parallel_AVC_PT

```
growTree (TreeNode root):
    spawn_root_and_wait (ProcessNodeTask (root))

ProcessNodeTask (TreeNode n):
  if (n is a leaf):
        return
    end if

    for each numerical attribute a ∈ A do:
      spawn (ProcessNumericalAttributeTask (a))
    end for
    for each categorical attribute a ∈ A do:
      spawn (ProcessCategoricalAttributeTask (a))
    end for
    wait_for_all

    Find the best splitting attribute a_split
    (node_ids, write_idx) = constructIndexArrays (a_split )

    for each attribute a ∈ A do:
      spawn (SplitAttributeTask (a, node_ids, write_idx))
    end for
    wait_for_all

    for each child node c do:
      spawn (ProcessNodeTask (c))
    end for
    wait_for_all
```

Algorithm 4.3.1 ProcessNumericalAttributeTask (a)

```
ProcessNumericalAttributeTask (a)
    for each block b do (in parallel):
        Partition b to get partition p
    end for

    for each partition do (in parallel):
        Construct and sort AVC-set from p
    end for

    step_size = 2
    while step_size <= n_threads do:
        for i = 0, step_size, step_size + step_size, ... do (in parallel):
            Merge AVC-set pair
        end for
        step_size = step_size * 2
    end while

    Evaluate a
```

Algorithm 4.3.2 ProcessCategoricalAttributeTask (a)

```
ProcessCategoricalAttributeTask (a)
    for each block b do (in parallel):
        Construct count matrix
    end for

    Aggregate count matrices
    Evaluate (a)
```

Algorithm 4.3.3 constructIndexArrays (a_{split})

```
constructIndexArrays (a_split)
    for each block b do (in parallel):
        update node_ids and write_idx
    end for
    return (node_ids, write_idx)
```

4.3 Evaluation

This section evaluates the performance of parallel_AVC, parallel_AVC_PD, parallel_AVC_PT and compares count-based algorithms to the sort-based ones.

4.3.1 Experimental Set-up and Test Data

We use three large datasets for our experiments, generated using a tool by [1]. The generator generates datasets with 6 numerical attributes, 3 categorical attributes and 2 classes. There are 10 generating functions and we choose to use function 1, 6, 7 (denoted as Func1, 6, 7 respectively). These functions have been used in [14] and produce trees with different sizes (see Table 4.2, 4.3 for tree features). Each dataset has 100 million records and the size of it is around 4GB. We use 90% of the data as training set and the remaining 10% as the testing set. Thus, the size of the actual training data is around 3.6GB with 90 million records. Experiments were run on a machine with an Intel Xeon CPU E5-2699 v4 @ 2.20GHz. It has Oracle Linux Server release 6.8 with kernel version 4.1.12-37.5.1.el6uek.x86_64 as the operating system. We used one socket and up to 16 physical cores with hyperthreading disabled.

Table 4.2: Features of trees created from different large datasets

Function	Tree size	Tree depth
Func1	5	2
Func6	45691	25
Func7	243770	26

Table 4.3: Distribution of the size of AVC-sets for Func1,6,7

Number of elements in an AVC-set	Number of attributes
0~10	2
10~100	0
100~1K	0
1K~10K	0
10K~100K	1
100K~1M	3

4.3.2 Results

Impact of $n\_partitions$

Before presenting results from different count-based algorithms, we first evaluate the impact of $n\_partitions$ for parallel_AVC_PD and parallel_AVC_PT. Figure 4.8-4.10 show this for all three Functions where $n\_partitions$ is increased from 16 to 256.

We first look at the results of the single-threaded versions. For Function 1, we can see that the two algorithms both perform better when there are more partitions. This is because our primary goal of using partitioning is to achieve cache efficiency when handling large datasets. The more the partitions, the fewer the elements each partition contains, and the more efficient the constructing and sorting of AVC-sets are. However, partitioning brings benefit for large nodes but not for small nodes. The original data for a small node usually fits in cache and thus can be efficiently sorted directly. This means that partitioning becomes an additional overhead when growing small nodes. This is why the trend does not exist for Function 6 and Function 7, where the tree contains many small nodes.

We next look at the parallel performance. The results show that for parallel_AVC_PD, larger $n\_partitions$ cause longer execution time, especially as the number of cores and the size of the tree increases. The reason is that for parallel_AVC_PD, the number of barriers depends on the number of partitions and the number of nodes in a tree. Recall that there are $(3 + n\_numerical \times log_2(n\_partitions)$ barriers per node, so there are more barriers when using more partitions and and growing larger trees. We can see that this is hardly an issue for parallel_AVC_PT, because there is no barriers in the algorithm. Thus, there is no penalty for using more partitions.

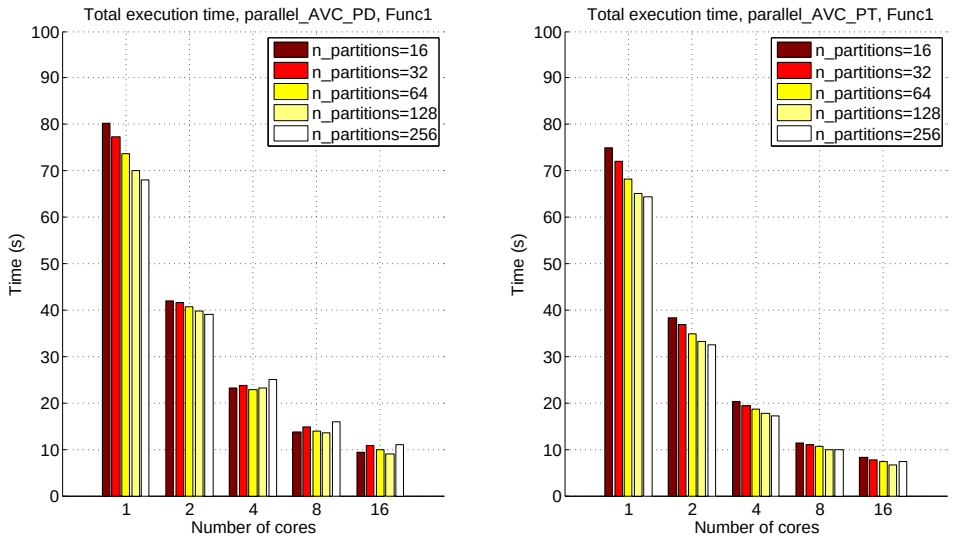


Figure 4.8: Impact of $n\_partitions$ on sequential and parallel performance, Func1

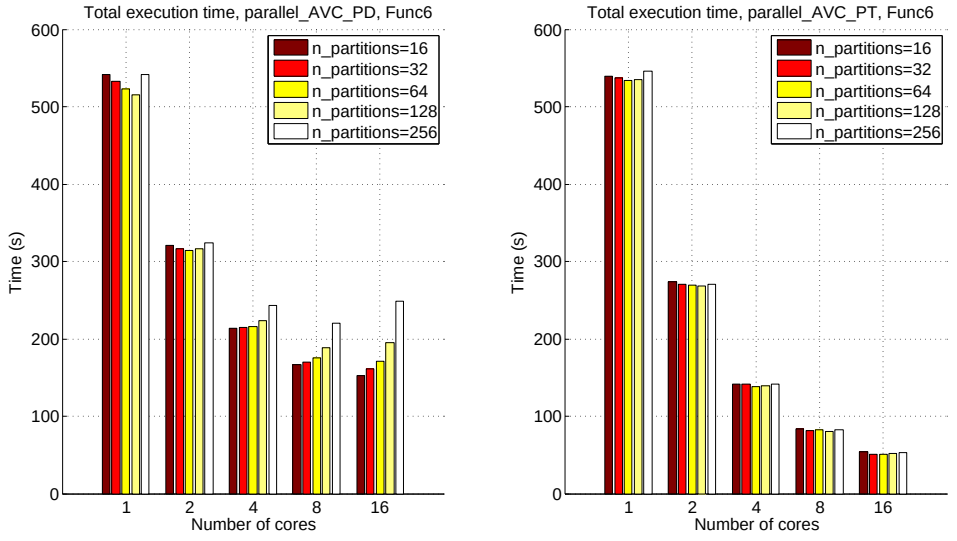


Figure 4.9: Impact of $n\_partitions$ on sequential and parallel performance, Func6

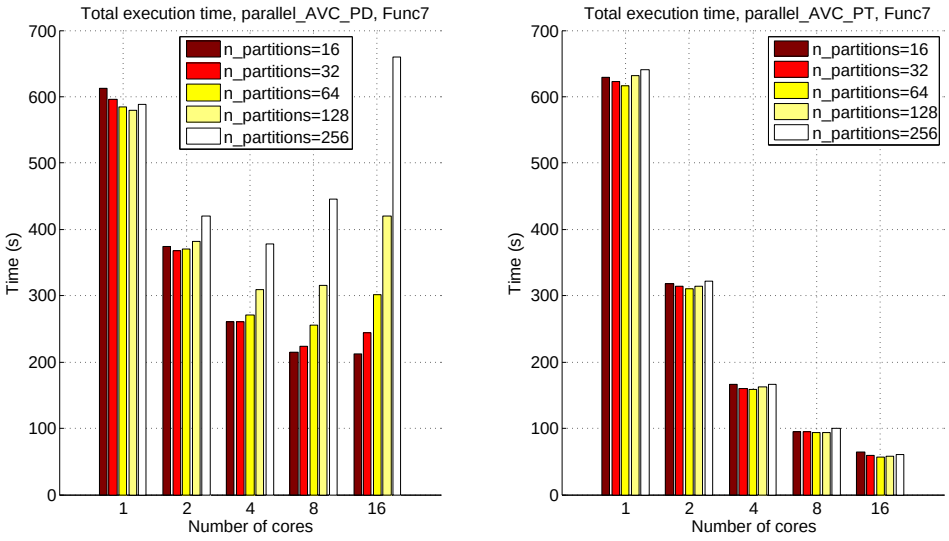
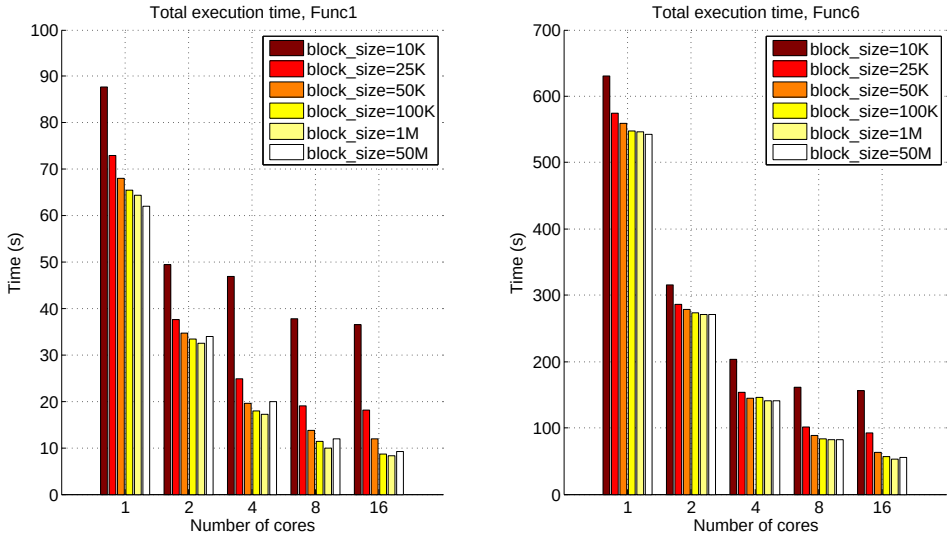


Figure 4.10: Impact of $n\_partitions$ on sequential and parallel performance, Func7

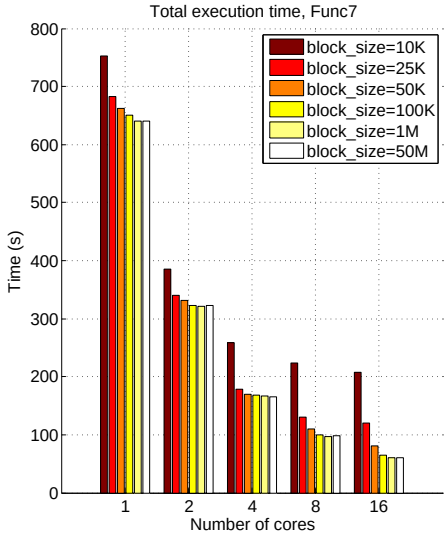
Impact of $block\_size$

We next examine how $block\_size$ affects the performance of parallel_AVC_PT. $n\_partitions$ was set to 256 for these tests. We present the results of experiments where $block\_size$ is equal to 10K, 25K, 50K, 100K, 1M and 60M. Results for these experiments are shown in Figure 4.11.

As can be expected, the execution time is longer when using small $block\_size$. For small $block\_size$, the granularity is finer and the overhead of task scheduling is greater. This is especially true when running on multiple cores since threads need to communicate to schedule tasks. On the other hand, if $block\_size$ is too large, we lose some parallelism when growing large nodes. This can be seen from the result of Function 1, where the tree is very small hence growing the tree is basically growing top-level large nodes. The difference is not so significant because there is still attribute-level parallelism. For Function 6 and Function 7, trees are large and growing large nodes no longer covers a large part of the total execution time. So even when $block\_size$ is large, the algorithm can still utilize other levels of parallelism to achieve good performance. We conclude that 1M is a good choice for $block\_size$.



(a)



(b)

Figure 4.11: Impact of $block\_size$ on parallel_AVC_PT

Comparison of count-based algorithms
We compare all three count-based algorithms. We set $n\_partitions = 16$ for parallel_AVC_PD and $n\_partitions = 128$, $block\_size = 1M$ for parallel_AVC_PT. These parameters provide the best scalability performance for the corresponding algorithm. The relative and total response time for each algorithm are shown in Figure 4.12-4.14.

Immediately obvious is how poorly the one-threaded parallel_AVC performs relative to the one-threaded version of the other two algorithms. parallel_AVC does not use partitioning, thus the cost of constructing and sorting large AVC-sets is large due to inefficient memory usage. This makes parallel_AVC an unattractive method despite that it scales to some extent with multiple cores.

We next focus on the parallel performance. parallel_AVC_PDs performance decreases as the number of nodes in the tree increases. This can be seen from the results of Function 6, where speedup ceases to increase at eight cores, and it is even more obvious for Function 7. This is as expected since synchronization becomes more of a significant factor of the overall response time when the tree gets larger. Also, whatever the number of partitions is, there will always be a fixed number of barriers per node for parallel_AVC_PD thus the scalability of parallel_AVC_PD is always limited.

Compared to parallel_AVC and parallel_AVC_PD, parallel_AVC_PT has shorter response time and scales fine. The drop in speedup on sixteen cores for Function 1 is due to the small size of the tree. parallel_AVC_PT employs all levels of parallelism (data level, attribute level, node level, task level) to the maximum extent. But for a small tree, there is a lack of node-level parallelism because there are only very few nodes. For this reason, load imbalance can happen at the end of growing a large node, and while there is not enough other nodes to work on, some threads have to wait and have nothing to do. Overall, we can conclude that parallel_AVC_PT can be used to classify large datasets.

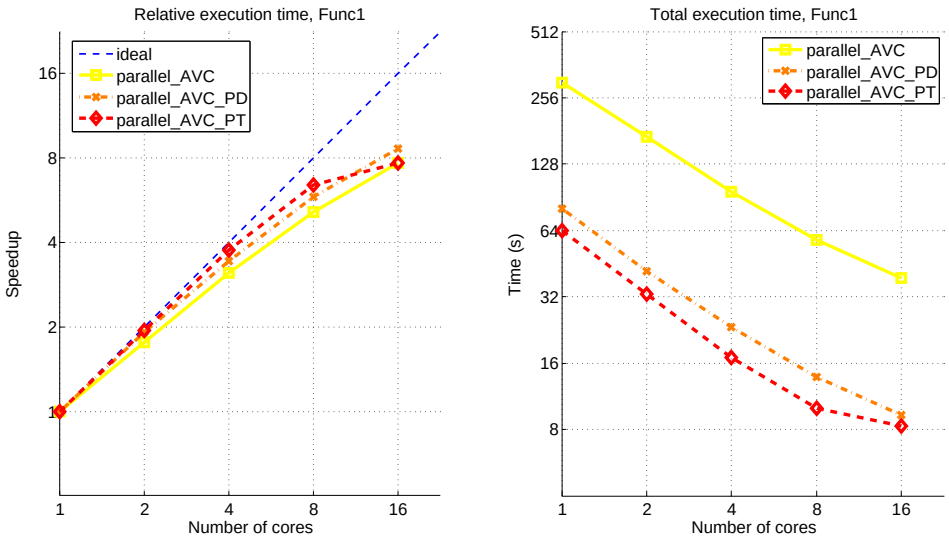


Figure 4.12: Speedup of count-based algorithms, Func1

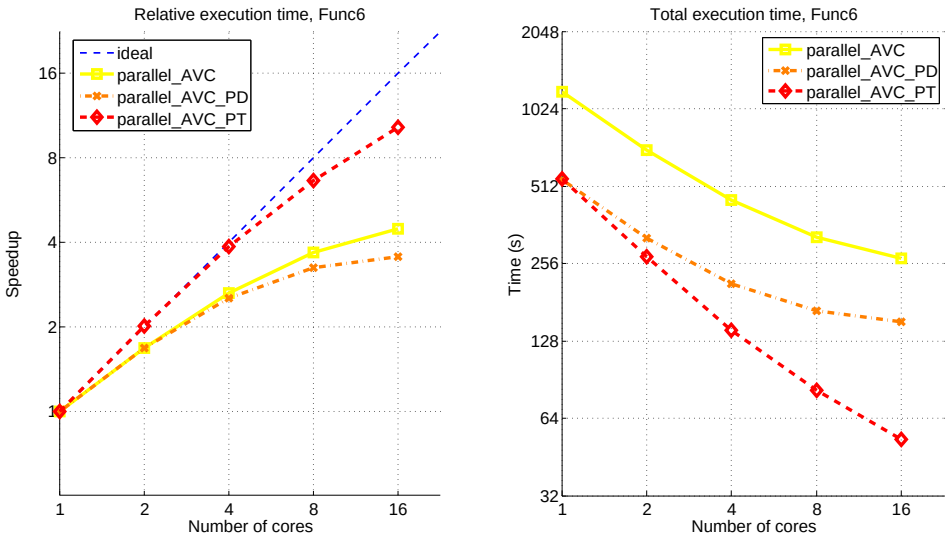


Figure 4.13: Speedup of count-based algorithms, Func6

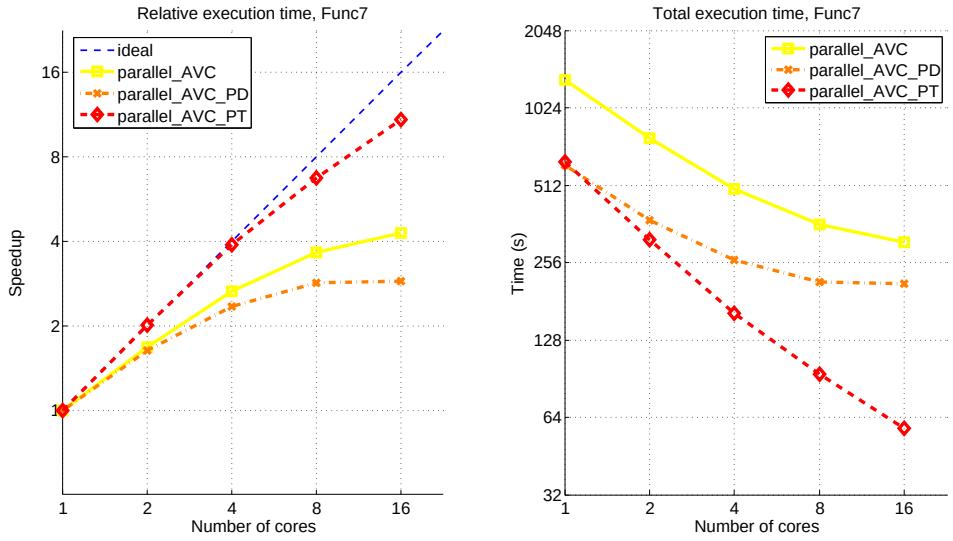


Figure 4.14: Speedup of count-based algorithms, Func7

Comparison with sort-based algorithm

Lastly, we give a comparison between the fastest count-based algorithm, parallel_AVC_PT and the fastest sort-based algorithm, parallel_AB. The pre-sorting of parallel_AB is parallelized by $\_\_gnu\_parallel :: sort()$[2].

We first compare parallel_AVC_PT with parallel_AB on the KDD dataset (Table 3.3). The experimental setup is the same as the one for the parallel algorithms in Section 3.4.1. The results in Figure 4.15 show that the two algorithms are competitive to each other on this smaller dataset.

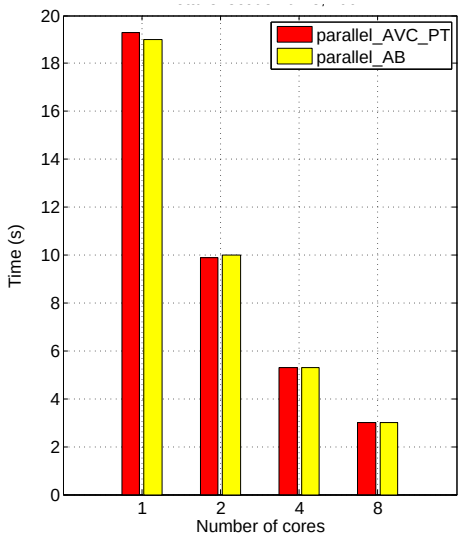


Figure 4.15: Total execution time of parallel_AB and parallel_AVC_PT on the KDD dataset

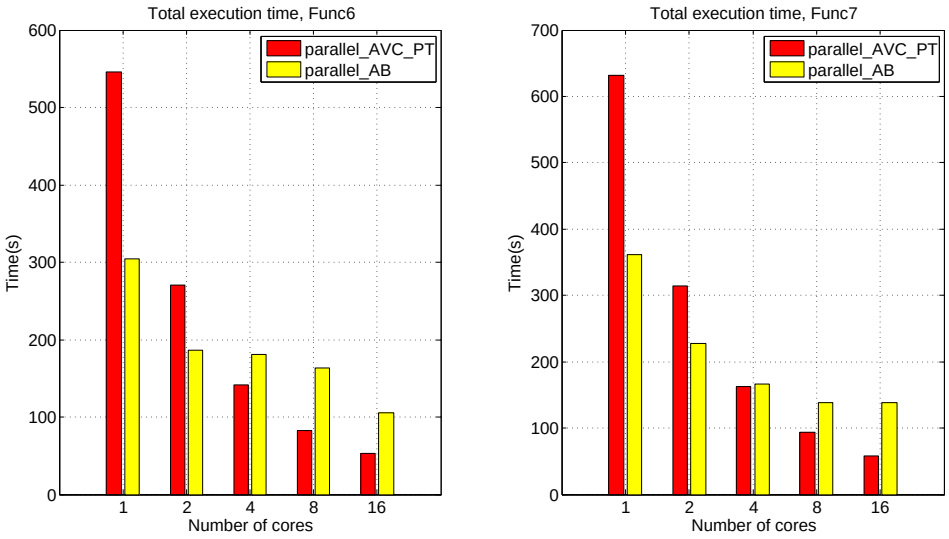
We next compare the two algorithms on the large datasets on 16 cores. The setup of this experiment is the same as Section 4.3.1 describes. Figure 4.16 show the total execution time of the two algorithms.

We initially focus on the single threaded versions. We see that parallel_AVC_PT is slower than parallel_AB when running on a single thread for Func6 and 7, but is faster for Func1. Recall that parallel_AVC_PT scans the data once to partition, and once to construct the AVC-sets from the partitions. It then sorts and merges the AVC-sets, and scans the AVC-sets to evaluate an attribute. In contrast, parallel_AB only scans the data once to evaluate an attribute. This means that compared to parallel_AVC_PT, it saves one scan over the data, the sorting and merging of the AVC-sets, as well as the final scan of the AVC-sets for each node. On the other hand, it requires pre-sorting and has a more expensive splitting phase. But the pre-sorting and the splitting phase are not as expensive as using partitioning and counting per node when there are a lot of nodes, in which case

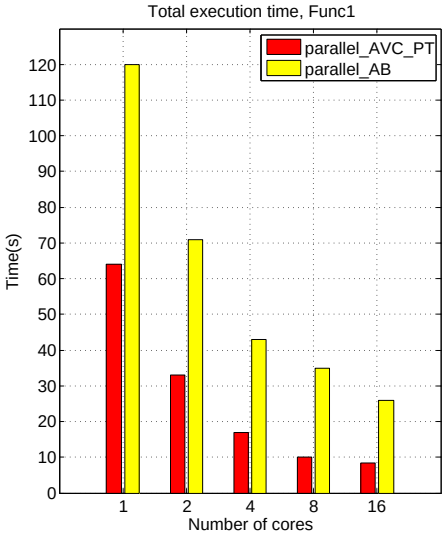
[2]https://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode.html.

we repeat the additional phases a lot of times and the cost of it overrides the benefit we get from partitioning and counting. But if there are only a few nodes, the count-based algorithm is still faster since there are not so much additional overhead and the cost of pre-sorting is more obvious for parallel_AB.

Although parallel_AVC_PT is slower on one thread for large trees, it is much better than parallel_AB in terms of scalability and raw performance when using multiple threads. As we can see, parallel_AB does not scale well for all three datasets. Note that the scalability shown here for parallel_AB is worse than the one shown in Figure 3.10. The reason for this could be that the sequential *Updating* phase for parallel_AB becomes more of a bottleneck when the dataset gets larger, for there are more records and it takes more time to update the position array. However, parallelizing it would have the same problem of random access since the position array is also randomly accessed in the *Updating* phase. And since random access would be a major bottleneck of the sort-based algorithms, we did not optimize our implementation further. Another reason for the difference in scalability of parallel_AB could be that the $node\_ids$ array is outside of cache more often for large datasets. However, more experiments need to be done to prove these statements.



(a)



(b)

Figure 4.16: Comparison of parallel_AVC_PT and parallel_AB

Chapter 5

Conclusion

We have implemented the C4.5 algorithm for decision tree induction, and parallelized it to run on multicore CPUs. We have investigated two types of algorithms, the sort-based algorithms and count-based algorithms.

The sort-based algorithm pre-sorts numerical attributes and keeps them in order throughout the growing process. We have demonstrated 4 ways to parallelize it but they all fail to scale well on eight-cores because keeping numerical attributes in order throughout the algorithm causes random access to the position array in the splitting phase, which in turn leads to cache inefficiency.

To improve cache efficiency, we took advantage of the AVC-sets which contain all the information for attribute evaluation but are only compact representations of the original data. This avoids pre-sorting and allows sequential access to the position array. In addition, we have combined the idea of AVC-sets with partitioning to further break large AVC-sets into small ones.

To reduce barriers and to best explore all potential parallelizations concurrently, we have employed the task parallelism scheme, which expresses tree building as tasks and allows different types of parallelization to be dynamically scheduled during the growing of the tree.

The count-based method seems promising, although it does not achieve completely linear speedup. One optimization would be that in the merging phase, instead of using a reduction tree and synchronizing on each level, one could merge two AVC-sets as soon as they are available. Another possible improvement would be to fuse the splitting of one node with the counting of its children, which could save the counting in the children. Future work can also be explored to port the algorithm to NUMA machines.

Bibliography

[1] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Database mining: A performance perspective. *TKDE*, 5(6):914–925, 1993.

[2] Marco Aldinucci, Salvatore Ruggieri, and Massimo Torquati. Decision tree building on multi-core using FastFlow. *CCPE*, 26(3):800–820, 2014.

[3] Nuno Amado, Joao Gama, and Fernando Silva. Parallel implementation of decision tree learning algorithms. In *Portuguese Conference on Artificial Intelligence*, pages 6–13. Springer, 2001.

[4] Yael Ben-Haim and Elad Tom-Tov. A streaming parallel decision tree algorithm. *Journal of Machine Learning Research*, 11(Feb):849–872, 2010.

[5] Borgelt. *DTree-Decision and Regression Tree Induction, Christian Borgelt's Web Page*, 2016.

[6] Chun-Chieh Chiu, Guo-Heng Luo, and Shyan-Ming Yuan. A decision tree using CUDA GPUs. In *iiWAS*, pages 399–402. ACM, 2011.

[7] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2009.

[8] Intel Corporation. *Intel Threading Building Blocks*, 2016. tbb2017_20160722oss.

[9] Wei Dai and Wei Ji. A Mapreduce Implementation of C4.5 Decision Tree Algorithm. *International Journal of Database Theory and Application*, 7(1):49–60, 2014.

[10] Johannes Gehrke, Venkatesh Ganti, Raghu Ramakrishnan, and Wei-Yin Loh. BOAT-Optimistic Decision Tree Construction. In *ACM SIGMOD Record*, volume 28, pages 169–180, 1999.

[11] Johannes Gehrke, Raghu Ramakrishnan, and Venkatesh Ganti. RainForest-A framework for fast decision tree construction of large datasets. In *VLDB*, volume 98, pages 416–427, 1998.

[12] Chris Giannella, Kun Liu, Todd Olsen, and Hillol Kargupta. Communication efficient construction of decision trees over heterogeneously distributed data. In *ICDM*, pages 67–74.

[13] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[14] Jin. Communication and memory efficient parallel decision tree construction. *SIAM*, 2003.

[15] Mahesh V Joshi, George Karypis, and Vipin Kumar. ScalParC: A new scalable and efficient parallel classification algorithm for mining large datasets. In *IPPS/SPDP*, pages 573–579, 1998.

[16] Roger J Lewis. An introduction to classification and regression tree (CART) analysis. In *SAEM*, pages 1–14, 2000.

[17] Xiao-Bai Li. A scalable decision tree system and its application in pattern recognition and intrusion detection. *Decision Support Systems*, 41(1):112–130, 2005.

[18] Seung Jae Lim, Cheolho Heo, Yunsik Hwang, and Taeseon Yoon. Analyzing Patterns of Various Avian Influenza Virus by Decision Tree. *International Journal of Computer Theory and Engineering*, 7(4):302, 2015.

[19] Win-Tsung Lo, Yue-Shan Chang, Ruey-Kai Sheu, Chun-Chieh Chiu, and Shyan-Ming Yuan. CUDT: a CUDA based decision tree algorithm. *The Scientific World Journal*, 2014.

[20] John MacCormick. How does the kinect work? *Presentert ved Dickinson College*, 6, 2011.

[21] Stefan Manegold, Peter Boncz, and Martin Kersten. Optimizing main-memory join on modern hardware. *TKDE*, 14(4):709–730, 2002.

[22] Manish Mehta, Rakesh Agrawal, and Jorma Rissanen. SLIQ: A fast scalable classifier for data mining. In *International Conference on Extending Database Technology*, pages 18–32. Springer, 1996.

[23] Ingo Müller. *Engineering Aggregation Operators for Relational In-Memory Database Systems*. PhD thesis, Karlsruher Institut für Technologie, 2016.

[24] Aziz Nasridinov, Yangsun Lee, and Young-Ho Park. Decision tree construction on GPU: ubiquitous parallel computing approach. *Computing*, 96(5):403–413, 2014.

[25] Biswanath Panda, Joshua S Herbach, Sugato Basu, and Roberto J Bayardo. Planet: massively parallel learning of tree ensembles with mapreduce. *VLDB*, 2(2):1426–1437, 2009.

[26] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[27] J Ross Quinlan. C4.5: Programming for machine learning. *Morgan Kauffmann*, page 38, 1993.

[28] Sanjay Ranka and V Singh. CLOUDS: A decision tree classifier for large datasets. In *KDD*, pages 2–8, 1998.

[29] Salvatore Ruggieri. Efficient C4.5 [classification algorithm]. *TKDE*, 14(2):438–444, 2002.

[30] Salvatore Ruggieri. YaDT: Yet another Decision Tree Builder. In *ICTAI*, pages 260–265, 2004.

[31] Fareena Saqib, Aindrik Dutta, Jim Plusquellic, Philip Ortiz, and Marios S Pattichis. Pipelined decision tree classification accelerator implementation in FPGA (DT-CAIF). *TC*, 64(1):280–285, 2015.

[32] John Shafer, Rakesh Agrawal, and Manish Mehta. SPRINT: A scalable parallel classi er for data mining. In *Proc. 1996 Int. Conf. Very Large Data Bases*, pages 544–555.

[33] Toby Sharp. Implementing decision trees and forests on a GPU. In *European conference on computer vision*, pages 595–608. Springer, 2008.

[34] D Strnad and A Nerat. Parallel construction of classification trees on a GPU. *CCPE*, 28(5):1417–1436, 2016.

[35] Mohammed Javeed Zaki, Ching-Tien Ho, and Rakesh Agrawal. Parallel classification for data mining on shared-memory multiprocessors. In *Data Engineering*, pages 198–205. IEEE, 1999.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

PERFORMANCE ANALYSIS OF DECISION TREE LEARNING ALGORITHMS ON MASSIVELY PARALLEL ARCHITECTURES

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
WANG	JINGYI

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
ZÜRICH, 18/11/2016	Jingyi Wang

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.