Exploiting Parallelism in Decision Tree Induction

Nuno Amado, João Gama, Fernando Silva

LIACC - University of Porto
R.Campo Alegre 823, 4150 Porto
{namado,jgama,fds}@ncc.up.pt

Abstract. In the fields of data mining and machine learning the amount of data available for building classifiers is growing very fast. Parallelism may be a good solution to reduce the amount of time spent in building classifiers from very-large datasets while keeping the classification accuracy. This work first overviews some strategies for implementing decision tree construction algorithms in parallel based on techniques such as task parallelism, data parallelism and hybrid parallelism. We then describe a new parallel implementation of the C4.5 decision tree construction algorithm using a breadth-first strategy, data and hybrid parallelism techniques. A novel contribution of this work is the ability to deal with missing values. Even though minor adjustments have still to be performed to the algorithm, we present some experimental results that can be used to predict its expected behaviour of the algorithm.

1 Introduction

Classification has been identified as an important problem in the areas of data mining and machine learning. Over the years different models for classification have been proposed, such as neural networks, statistical models as linear and quadratic discriminants, decision trees, and genetic algorithms. Among these models, decision trees are particularly well suited for data mining and machine learning. Decision trees are relatively faster to build and achieve similar, sometimes better, accuracy when compared with other classification methods [7, 9]. Nowadays, the growth of transactional data being stored in servers is exponential. It is, therefore, important to have classification algorithms computationally efficient and scalable. Parallelism may be a solution to reduce the amount of time spent in building decision trees using larger datasets while keeping high classification accuracy levels [4, 5, 11, 12]. Parallelism can be easily achieved by building the decision tree nodes in parallel or by distributing the training data among the computing elements. However, implementing parallel algorithms for building decision trees is a complex task for the following reasons. First, the shape of the tree is highly irregular, it is determined only at runtime, and the amount of processing required at each node may vary significantly. Hence, any static allocation scheme will probably suffer from a high load imbalance problem. Second, even if the successors of a node are processed in parallel, their
construction requires sharing part of the data of their parent node. If the data is dynamically distributed by the processors which are responsible for different nodes then it is necessary to implement a strategy for data movement. If the data is not properly distributed then the performance of the algorithm may degrade due to a loss of locality [5, 12].

Decision trees are usually built in two phases: tree construction and simplification phases. With no doubt, the most computational expensive phase of the algorithm is the tree construction phase[2, 9, 10, 12]. Therefore, in this work, for the description of the parallel construction tree algorithms we only consider the first phase.

The main contribution of this work is the design and parallel implementation of the C4.5 decision tree algorithm. Our implementation of the decision tree follows a breadth-first strategy that we find more appropriate for parallelisation. Continuous attributes are sorted only once before starting growing the decision tree. This bypasses a known source of overhead, also found in the normal C4.5 strategy, which is the need to sort the continuous attributes at each decision node. Our implementation also deals with missing values even though this increases the implementation complexity. The reason for this is that when dealing with an example with a missing attribute value for the test attribute at a decision node, it must be propagated for all descendant branches of that node.

In the remainder of the paper we first overview some strategies for implementing parallel decision tree algorithms, and then describe our parallel implementation of the C4.5 decision tree construction algorithm. We then present some experimental results and draw some conclusions about the expected behaviour of the algorithm.

2 Related work

This section overviews some strategies for implementing decision tree construction algorithms in parallel using task parallelism, data parallelism and hybrid parallelism.

Task parallelism. The construction of decision trees in parallel by following a task-parallelism approach can be viewed as dynamically distributing the decision nodes among the processors for further expansion. A single processor using all the training set starts the construction phase. When the number of decision nodes equals the number of processors the nodes are split among them. At this point each processor proceeds with the construction of the decision sub-trees rooted at the nodes of its assignment.

This approach suffers, in general, from bad load balancing due to the possible different sizes of the trees constructed by each processor. To avoid excessive communication overheads, the implementations presented in [4] require the whole training set to be replicated in the memory of all the processors.

Data parallelism. The use of data parallelism in the design of parallel decision tree construction algorithms can be generally described as the execution
of the same set of instructions (algorithm) by all processors involved. The parallelism is achieved by distributing the training set among the processors where each processor is responsible for a distinct part of the data. The distribution of the data can be performed in two different ways, horizontally or vertically.

The parallel strategy based on vertical data distribution [5] consists in splitting the data by assigning a distinct set of attributes to each processor. Each processor keeps in its memory only the whole values for the set of attributes assigned to it and the values of the classes. During the evaluation of the possible splits each processor is responsible only for the evaluation of its attributes.

Parallelism with vertical data distribution can still suffer from load imbalance due to the evaluation of continuous attributes which requires more processing than the evaluation of discrete attributes. However, this can simply be avoided by distributing the continuous attributes evenly among the processors. This approach to parallelism presents poor scalability. As the number of attributes describing the data are usually not very high, when the number of processors grows, the number of attributes in each process decreases and the time spent communicating overcomes the time spent in processing the data.

The parallel strategy based on horizontal data distribution consists of distributing the examples evenly by the processors. Each processor keeps in its memory only a distinct subset of examples of the training set. The possible splits of the examples associated to a node are evaluated by all the processors, which communicate between them to find the global values of the criteria used and, by this, the best split. Each processor performs locally the split. In [11], the authors describe the implementation of two algorithms with horizontal data distribution: SLIQ and SPRINT. For the evaluation of a split based on a continuous attribute, these systems have the set of examples sorted by the values of the attribute. In order to avoid sorting the examples every time a continuous attribute is evaluated, they use separate lists of values for each attribute, which are sorted once at the beginning of the tree construction. SLIQ also uses a special list, called the class list, which has the values of the class for each example. For efficiency reasons, this class list stays resident in memory during all the tree construction process. In the parallel implementation of SLIQ the training set is distributed horizontally among the processors where each processor is responsible for creating its own attribute lists. In SPRINT, the class list is eliminated by adding the class label in the attribute lists entries. The index in the attribute lists is now the index of the example in the training set. The evaluation of the possible splits is also performed by all processors, which communicate between them to find the best split. After finding the best split, each processor is responsible for splitting its own attribute lists.

Both systems report good performance and scalability results, being also capable of processing very-large datasets. However, the operation to perform the split of the set of examples associated with a node requires high communication load in both systems.

**Hybrid parallelism.** The parallel decision tree construction algorithms, which use hybrid parallelism, can be characterised as using both data parallelism
with horizontal or vertical distribution and task parallelism. The implementation of hybrid parallel algorithms is strongly motivated by the choice between the distribution of the amount of processing at each node and the required volume of communications. For the nodes covering a significant amount of examples, it is used data parallelism to avoid the problems already stated of load imbalance and of poor use of parallelism associated with task parallelism. But, for the nodes covering fewer examples the time used for communications can be higher than the time spent in processing the examples. To avoid this problem, when the number of examples associated to a node is lower than a specific value, one of the processors continues alone the construction of the tree rooted at the node (task parallelism). Usually, the switch between data parallelism and task parallelism is performed when the communications cost overcomes the processing and data transfer costs. In general, hybrid parallelism approaches achieve better speed-up and efficiency results than the other approaches. A parallel decision tree construction algorithm using hybrid parallelism is described in [6].

3 C4.5 Parallel Implementation

In this section, we describe a new parallel implementation of the C4.5 decision tree construction algorithm. This implementation follows a hybrid parallelism strategy with the use of data parallelism at the beginning of the decision tree build process and task parallelism at the lower nodes of the tree which cover a smaller amount of examples.

Data parallelism is implemented using an horizontal data parallelism strategy similar to the one used by the parallel implementation of SLIQ [11]. Our implementation is designed to be executed in a distributed memory environment where each of the $k$ processors has its own private memory. It addresses two fundamental issues: load balance and locality. A uniform load balance is achieved by distributing horizontally the examples equally among the $k$ processors and using a new breath-first strategy to build the decision tree. When the number of examples in every node yet to explore is lower than a predefined threshold, the nodes are distributed evenly among the processors, along with the examples covered by them, and, each process, continues to explore its nodes independently. To avoid load imbalance, characteristic of task parallelism, when a process finishes exploring its nodes, sends a request for more nodes to the other processors.

Data Parallelism. As in the parallel version of SLIQ, each processor is responsible for building its own attribute lists and class lists from the subset of examples assigned to it. The entries in the class lists keep, for each example, the class label, the weight (used in the C4.5 algorithm to deal with unknown values), the corresponding global index of the example in the training set and a pointer to the node in the tree to which the example belongs. The attribute lists also have an entry for each example with the attribute value and an index pointing to the example corresponding entry in the class list. The continuous attribute lists are globally sorted by the values of the attribute using the sorting algorithm described in [3]. Each processor has now, for each continuous attribute, a list of
sorted values where the first processor has the lower values of the attribute, the second has the attribute values higher than the first processor and lower than the third, and so on. After the sort, each processor updates its class list with the examples information corresponding to the new values of the continuous attributes that were not initially assigned to it.

The main problems in the parallel tree construction process reside in performing the split and in finding the best split for the set of examples covered by a node. For these two steps, communication among processors is necessary in order to determine the best global split and the assignments of examples to the new subsets resulting from the split.

In the parallel version of the algorithm each processor has a distinct subset of values of the continuous attributes in their globally sorted lists. Before each processor starts evaluating the possible splits of the examples, the distributions must be initialised to reflect the examples assigned to the other processors. Each processor finds its distributions of the local set of examples for each attribute and sends them to the other processors.

For the nominal attributes, these distributions are gathered in all processors which locally compute their gain. For continuous attributes the gain of a possible split, is found based upon the distributions before and after the split point. When a processor receives the distributions from another it initialises, for each continuous attribute, the distributions before the split point with the distributions of the processors with lower rank, and the distributions after the split point are initialised with those from the processors with higher rank. After evaluating all possible divisions of their local set of examples, the processors communicate among themselves in order to find the best split from the $k$ local best splits found by each one.

As soon as the best split is found, the split is performed by creating the child nodes and dividing the set of examples covered by the node. This step requires that each processor updates the pointers to the nodes in their class lists and divide the attribute lists in many lists, as a result of the split, assigning them to the new child nodes. For the split each processor scans its local list of the attribute chosen and, depending on the value of the attribute, each entry is moved to the corresponding sublist, and the pointer to the node in the class list is updated.

In order to divide the remaining attribute lists, the pointers to the nodes in the class list are used. But, before that, the class lists in each process must be updated. The entries in the class list of one processor, whose values for the chosen attribute are kept in another processor, must be updated with the information of the node to which the corresponding examples were assigned by the other processor. For each example in the set covered by the node, the index, weight and the node to which they were assigned are gathered in all processors allowing each processor to update its local class list and divide the attribute lists. When dividing the attribute lists, each processor finds its local class distributions for each attribute in the new sets of examples assigned to the child nodes. Still in
this phase, the distributions are scattered allowing for each processor to know
the global class distributions used later during the splits evaluation phase.

**Task Parallelism.** The decision tree is built using horizontal data parallel-
ism while the amount of examples covered by the nodes stays above a pre-
defined threshold. When all nodes yet to explore cover a number of examples
smaller then this threshold they are distributed evenly among the processors
along with the examples covered by them. As in any hybrid parallelism ap-
proach the criteria used to switch from data parallelism to task parallelism is
one of the most important factors for the performance of the algorithm [6]. If
the switch between the two strategies is performed too early the algorithm will
suffer from bad load balance, characteristic of task parallelism, or, if it is per-
formed too late, it will suffer from high communication costs at the nodes with
less examples, characteristic of data parallelism.

In our algorithm, the switch between the two strategies is performed when-
ever the total communications cost for building a node using data parallelism
overcomes the cost of building the node locally at one process plus the cost of
moving the set of examples covered by the node between processors. As already
stated, in order to avoid poor load balance, when a process finishes exploring all
its nodes it sends a request for new nodes to the other processors.

Our parallel decision tree algorithm preserves most of the main features of
C4.5. More importantly, it deals with difficult issues such as the support of
missing attribute values. Furthermore, since the same evaluation criteria were
considered in the parallel version, it obtains the same splitting tests, hence leading
to the same decision tree and classification results as those obtained by C4.5. We
also use different data structures to achieve parallelism, namely attribute lists
and class lists as used in the SLIQ strategy. The main reason for using separate
lists for the attributes is to avoid sorting the examples every time a continuous
attribute is evaluated. The several sorts performed by C4.5 during the evaluation
of the attributes are one of its efficiency limitations [2, 10], and, if they were
kept in the parallel version then they would strongly limit the performance of
the algorithm. Hence, by using separate lists for the attributes makes it possible
to globally sort the continuous attributes only once at the beginning of the tree
construction process.

The SPRINT strategy was developed to overcome the use of centralised struc-
tures such as the class list. It avoids the class lists by extending the attribute
lists with two extra fields, the class label and the global index of the example in
the training set. The main disadvantage stated in [11] for the use of the class list
is that it can limit the size of the training set that can be used with the algo-
rithm, due to the fact that it must stay resident in memory during the execution
of the algorithm. Suppose now that we have a training set with $N$ examples
and $A$ attributes and each field of the attribute lists entries require one word of
memory. Suppose also that we have $k$ processors available for the execution of
the algorithms. In this situation, SPRINT will need $3 \times A \times N/k$ memory words in
each processor to store the training set divided equally among the $k$ processors.
Our algorithm requires that in the worst case, when performing the global sort
of the continuous attribute lists, the class list must store the information for all examples in the training set. In this situation the amount of memory necessary to store the training set in each processor will be $2 \times A \times N/k + 3 \times N$ memory words, resulting respectively from the attribute lists and class lists. It is easy to observe that when the number of attributes is three times greater than the number of available processors ($A > 3 \times k$) it is better to use, in terms of memory, the class list than the attribute lists.

Another disadvantage of using class lists, stated in [11], is the time required to update it during the split. In the implementation of SLIQ, the class list is entirely replicated in the memory of each processor. After the split each processor has to update the entire list, which limits the performance of the algorithm. Similarly to SLIQ and SPRINT, in our parallel version of C4.5, to perform the split the processors exchange the global indexes of the examples covered by a node with the information to which child node they were assigned. Each processor keeps a list of pointers to the entries of the class list sorted by the examples indexes in order to improve the searches when updating the class list with the information from the other processors.

4 Experimental results

In this section we present some experimental results obtained with the data parallelism approach of our algorithm. Although these results refer only to the behaviour of the data parallelism approach, they can be used to predict the expected behaviour of the whole algorithm.

As mentioned before, our decision tree algorithm uses the same criteria and procedures used in the C4.5 algorithm. Therefore, they produce the same decision trees with equal classification accuracy. Consequently, we evaluate the performance of our system only in terms of execution time required for the construction of the decision tree. Again, the prune phase of the algorithm was not taken into account for the time measured in the experiments as it is negligible.

The parallel version of C4.5 was implemented using the standard MPI communication primitives [8]. The use of standard MPI primitives allows the implementation of the algorithm to be highly portable to clusters of machines. The experiments presented here were conducted on a PC server with four Pentium Pro CPUs and 256MB of memory running Linux 2.2.12 from the standard Red-Hat Linux release 7.2 Distribution. Each CPU in the PC server runs at 200MHz and contains 256Kb of cache memory. All the time results presented are the average result of three executions of the algorithm.

All experiments used the Synthetic data set from Agrawal et al [1]. This dataset has been previously used for testing parallel decision tree construction algorithms such as the systems SLIQ and SPRINT [11, 12]. Each example in the dataset has 9 attributes, where 5 of them are continuous and 3 are discrete. The values of the attributes are randomly generated. Agrawal et. al. also describe 10 classification functions based on the values of these attributes and with different
complexities. For the datasets used in the experiments it was only considered one of those functions, named Function2.

The first set of experiments measured the performance, in terms of speedup, of the algorithm. For each experiment we kept constant the total size of the training set while the number of processors varied from 1 to 4. Figure 1 illustrates the speedup results for training sets of 100k, 200k and 400k examples. Our algorithm overall shows good speedup results and, as expected, the best results were obtained for the largest dataset used (with 400K examples). The slowdown observed for the largest processor configuration is mainly due to the high communication costs in the split phase.

The next set of experiments aimed to test the scale-up characteristics of the algorithm. In these experiments, for each scale-up measure, the size of the training set in each processor was kept constant (50k, 75k and 100k examples) while the number of processors varied from 1 to 4. Figure 2 shows that our algorithm achieves good scale-up behaviour. Again, high communication overheads in the split phase influence scale-up capacity as the number of processors increases.
5 Conclusion

In this work we overviewed some parallel algorithms for constructing decision trees and proposed a new parallel implementation of the C4.5 decision tree construction algorithm using a breadth-first strategy. The main features of C4.5, such as the ability of dealing with missing attribute values, were preserved. As a result, our system builds the same decision trees, as those obtained by C4.5, with the same classification accuracy.

The preliminary results showed that the algorithm achieves good speedup and scale-up performance indicating that it can be used to build decision trees from very-large datasets. The results also indicate that communication overheads are a key factor in limiting the performance of the algorithm, specially when each process has a relatively small set of examples. Furthermore, the results indicate that the use of task parallelism for building the nodes covering a small set of examples will lead to better performance results since the communications overhead will be significantly reduced.

Acknowledgements

The authors are grateful to the anonymous referees for their valuable comments. The work presented in this paper has been partially supported by projects APRIL (POSI/SRI/40749/2001), ALES (POSI/SRI/39770/2001) and funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSI.

References
