Parallel Implementation of Decision Tree Algorithm.

Abdur Rehman Raja and Usman Saeed

DS8001

December 21, 2017

1. Abstract.

In the disciplines of data mining and machine learning the amount of data usable for building classifiers is growing really fast. Thus there is a big demand for algorithms that are capable of building classifiers from very-large datasets and, simultaneously, being computationally efficient and scalable. One potential answer is to employ parallelism to reduce the sum of time expended in building classifiers from very-large datasets and keeping the classification accuracy [1]. Our work for this project first overviews strategy for implementing decision tree by generating a univariate data and then implementing classification tree algorithm. Later one there are few suggestions made to parallelize algorithm and then we implemented some techniques to parallelize same algorithm and generate the results for the scope of this project.

2. Background.

Classification has been identified as an important problem in the areas of data mining and machine learning. Over the year's 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 suited for data mining and machine learning. Decision trees are relatively faster to build and obtain similar, sometimes higher, accuracy when compared with other classification methods [1].

Nowadays there is an exponential growing on the data stored in computers. Therefore, it is 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 [1] . Parallelism can be easily achieved by building the tree decision nodes in parallel or by distributing the training data. However, implementing parallel algorithms for building decision trees is a complex task due to the following reasons. First, the shape of the tree is highly irregular and it is determined only at runtime, beyond the fact that the amount of processing used for each node varies. 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 needs part of the data associated to the parent. If the data is dynamically distributed by the processors which are responsible for different nodes, then it is necessary to implement data movements. If the data is not properly distributed, then the performance of the algorithm can suffer due to a loss of locality [1].

Univariate decision trees are one of the most widely used classification model in data mining. First the ID3 algorithm is based on discrete features appeared then in C4.5 (Quinlan, 1993) it is expanded to include continuous features. Constructing a univariate decision tree has time complexity of roughly O(dfNlogN), where N is the total sample size, f is the number of features and d is number of nodes in the tree. In data mining applications, the sample size tends to be very large. So constructing decision trees in parallel manner becomes an important fact [2].

3. Widely used Techniques.

There have been two major categories of attempts

Attribute-parallel: Data are vertically partitioned according to the attributes and allocated to different machines, and then in each iteration, the machines work on non-overlapping sets of attributes in parallel in order to find the best attribute and its split point (suppose this best attribute locates at the i-th machine. This process is computationally very efficient. However, after that, the re-partition of the data on other machines than the i-th machine will induce very high communication costs (proportional to the number

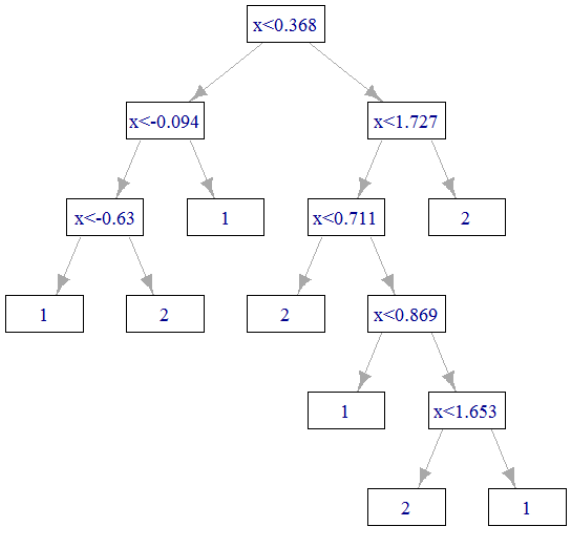
of data samples). This is because those machines have no information about the best attribute at all, and in order to fulfill the re-partitioning, they must retrieve the partition information of every data sample from the i-th machine. Furthermore, as each worker still has full sample set, the partition process is not parallelized, which slows down the algorithm [3].

Data-parallel: Training data are horizontally partitioned according to the samples and allocated to different machines. Then the machines communicate with each other the local histograms of all attributes (according to their own data samples) in order to obtain the global attribute distributions and identify the best attribute and split point. It is clear that the corresponding communication cost is very high and proportional to the total number of attributes and histogram size. To reduce the cost, we can exchange quantized histograms between machines when estimating the global attribute distributions. However, this does not really solve the problem – the communication cost is still proportional to the total number of attributes, not to mention that the quantization may hurt the accuracy [3].

4. Methodology.

Detail about creation of sequential decision tree.

The base of the algorithm is an entropy function (‘get_entropy’). It gets values of X and classes C for each value of X. Lower the value of entropy the better. For example, if we have values of X only class=1 the entropy = 0 and there is no need for classification (there are no other classes in training set). We will have the maximum value of an entropy if we have equal number of classes 1 and 2. The idea of a decision tree consists to split training data (X, C) using minimum split entropy. For calculation of split entropy, we use 'get_split_entropy'. It gets X and C, splits X and C into two parts by node, estimates for each part the values of entropy and calculates split entropy. The node is position at some value of X. For finding the minimum split entropy is applied, a function 'fnd_node'. The function finds the best position for splitting X and C into two parts.
For creation of the decision tree, we use a function 'crt_dtree'. In this function we split the initial training set (X,C) by minimum split entropy and add new sub-training sets (X1,C1) and (X2,C2) in the end of list. Then we test entropy (X1,C1). If it is equal to 0 the sub-training set is the leaf, we save the information. Else, we split (X1,C1) by minimum split entropy and add new sub-sub-training sets (X11,C11) and (X12,C12) in the end of list. Then we test entropy for (X2,C2) and perform same thing again. Function stops the cycle when its left with only leafs.



For example, we get the new the value of X is equal 1.2. Algorithm will check which class is this value. We start from the top of the tree. It checks if *"the value < 0.368"?* answer is yes so it goes to the right. *value < 1.727?* answer is no then it goes to the left. *value <0.711*? answer is No so it goes to the right…and so on. This is assuming we have 2-classes for the value.

We used random number generators to create our dataset with 2 classes as their labels. The algorithm was tested on 10, 100, 1000, 100000 and 1 million rows to see the difference between each dataset.

4.1: Attribute-Parallel

The portion we focused mainly on was parallelizing the 'fnd_node' function. This function finds the node that has the minimum entropy and parallelizes the calculation of finding the minimum entropy for each node. It does this by finding all the entropy values for each node then reduces the values by selecting the minimum entropy value. This method parallelizes the 'get_entropy' and 'get_split_entropy' function as well because both of these functions are involved when calculating the minimum entropy.

5. Results

Table 1: Sequential Results

Dataset	Run Time (sec)
10 Random Numbers	0.0010
100 Random Numbers	0.056
1000 Random Numbers	0.122
100,000 Random Numbers	4580
1,000,000 Random Numbers	18 hours

Table 2: Experiment Results

Dataset	Run Time (sec)
10 Random Numbers	0.0015
100 Random Numbers	0.071
1000 Random Numbers	0.232
100,000 Random Numbers	2780
1,000,000 Random Numbers	13 hours

The above shows the run time of parallel implementation for different dataset sizes. The sequential method doesn't scale well for large datasets like the 100,000 row dataset and the million row dataset. While this is the case for sequential, the results for parallel implementation for the 10, 100 and 1000 rows performed worse than the sequential datasets. This could be because of the overhead associated with the implementation which outweighs the actual benefit of the implementation

6. Summary

We presented the decision tree and implemented an attribute-parallel implementation of calculating entropy. Although our results didn't perform well with regards to smaller datasets, the implementation performed very well on the larger datasets. We partially achieved our original goal of decreasing run-time. Nowadays most datasets have a larger size than a 1000 rows so in that sense it doesn't really make much of difference if our implementation performed worse on small datasets. Our main goal was to prove that parallelization is effective in general.

References

[1] Nuno Amado, Jo˜ao Gama, and Fernando Silva, *Parallel Implementation of Decision Tree Learning Algorithms In: Progress in Artificial Intelligence pp 6-13*

[2] Yıldız, O. T., O. Dikmen, "Parallel Univariate Decision Trees", Pattern Recognition Letters, Vol 28, No 7, pp. 825-832, 2007

[3] Qi Meng, Guolin Ke, Taifeng Wang, Wei Chen, Qiwei Ye, Zhi-Ming Ma, and Tie-Yan Liu, A Communication-efficient Parallel Algorithms for Decision Tree, Advances in Neural Information Processing Systems 30 (NIPS), 2016.