Parallel Univariate Decision Trees

Olcay Taner Yıldız, Onur Dikmen

Technical Report, FBE/COE-02/2004-13

Institute of Graduate Studies in Science and Engineering

Department of Computer Engineering

Boğaziçi University

yildizol@cmpe.boun.edu.tr

onuro@boun.edu.tr

August 31, 2004

Acknowledgements

We thank to Burak Gürdağ for fruitful discussions on parallel computing and his help in using Asma. We would like to thank Prof. Ethem Alpaydn for his support and advises.

ABSTRACT

Univariate decision tree algorithms are widely used in Data Mining because (i) they are easy to learn (ii) when trained they can be expressed in rule based manner. In several applications mainly including Data Mining, the dataset to be learned is very large. In those cases it is highly desirable to construct univariate decision trees in reasonable time. This may be accomplished by parallelizing univariate decision tree algorithms. In this paper, we first present two different univariate decision tree algorithms C4.5 and univariate Linear Discriminant Tree. We show how to parallelize these algorithms in three ways: (i) feature based, (ii) node based (iii) data based manners. Experimental results show that performance of the parallelizations highly depend on the dataset and the node based parallelization demonstrate good speedups.

1. Introduction

Univariate Decision trees are one of the most widely used classification model in Data Mining. First ID3 algorithm based on discrete features appeared in [9], then in C4.5[10] it is expanded to include continuous features. Constructing a univariate decision tree has time complexity of roughly $\mathcal{O}(dfNlogN)$, 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 became an important fact.

Parallel construction of univariate decision trees can be divided into two groups. Proposing parallel decision tree algorithms and parallel formulations of existing algoritms.

SLIQ[6] is a unvariate parallel decision tree classifier that can handle both numeric and continuous attributes. By using a pre-sorting technique in a breadth-first tree-growing phase, it is able to classify disk-resident datasets. SLIQ also introduced a new tree-pruning algorithm. Since pruning phase of the decision trees takes much smaller time than tree growing phase, we will not consider the time used in tree pruning in our discussions.

SPRINT[11] extends the idea of SLIQ by addressing the problems with it. Although SLIQ uses the entire dataset to build the tree, it requires data, which is directly proportional to the number of features in the dataset to stay in memory all the time. This limits the amount of data that can be classified by SLIQ. SPRINT is intended to remove these memory restrictions.

Kufrin[5] proposed a data distributed parallel formulation of C4.5 algorithm. He mentioned that, since we only use sorting to gather the frequency statistics from data, we can also perform concurrent sorts on each processor. Frequency statistics for each local candidate split point is evaluated and shared with other processors to get the best

split at each node.

Srivastava et al.[12] described two parallel formulations in their paper. Synchronous Tree Construction Approach and Partitioned Tree Construction Approach. They discuss the advantages and disadvantages of these two approaches and propose a hybrid methodology. In syncronous tree construction approach, all processors construct a decision tree synchronously by sending and receiving class distribution information of local data. In partitioned tree construction approach, different processors work on different parts of the classification tree, whenever feasible. The hybrid approach they proposed, starts and continues with the first approach as long as the communication cost of the approach is not high. Once this cost is high they switch to the second approach.

In this paper we give and compare the performances of the three different formulations of two different univariate decision tree construction algorithms. C4.5 and the univariate version of Linear Discriminant Trees[13]. In Section 2 we give serial formulations of these two algorithms. Three different parallel formulations are explained in Section 3. Hardware and software platforms used in the experiments and the descriptions of the datasets are detailed in Section 4. Section 5 presents experiments and discussion on the results of these experiments.

2. Serial Formulations

In this section, we explain two univariate decision tree algorithms C4.5 and LDT. Since each algorithm does the same job at each decision node, we give the algorithms for a single node n. Each node works on an instance space of x. Each instance of the space x has f features, which can be discrete or continuous.

If the best split is found at node n, both algorithms create two or more child nodes. The instances of node n are also distributed to child nodes according to the best split. Figure 2.1 gives the pseudocode of this child node creation.

```
CREATE_CHILDREN(Node n, Instances x, bestfeature)
    if bestfeature discrete
        Partition instances x into k groups, k is the
        number of all possible values of the bestfeature
        Split node n into k child nodes
    endif
    if bestfeature continuous
        Partition instances x into 2 groups
        Split node n into 2 child nodes
    endif
```

Figure 2.1. Creating Child Nodes

2.1. C4.5

Original serial C4.5 algorithm is given in Figure 2.2. Finding best split differs in discrete and continuous features. There is one possible split for discrete feature, whereas in continuous features there are as many split points as the number of samples in that node. By sorting with respect to the values of the instances for that feature, we evaluate all possible split points in one pass. Since sorting values takes $\mathcal{O}(NlogN)$ by using Quicksort algorithm and making one pass to evaluate all possible split points takes $\mathcal{O}(N)$ time, the bottleneck of the C4.5 algorithm is the sorting phase.

```
C45(Node n, Instances x, Features f)
    for each feature i in f
        if f_i discrete
            Calculate information gain g_i
            if g_i < bestgain
                bestgain = g_i
                bestfeature = i
            endif
        endif
        if f_i continuous
            Sort x
            for each different value j of f_i
                Calculate information gain g_ij
                if g_ij <bestgain
                    bestgain = g_ij
                    bestfeature = i
                endif
            endfor
        endif
    endfor
    CREATE_CHILDREN(n, x, bestfeature)
    Call C45 for each child node
```

Figure 2.2. C4.5 Algorithm

2.2. LDT

Pseudocode of the LDT algorithm is given in Figure 2.3. This algorithm works the same as C4.5 for discrete features. For continuous features, finding the best split via Fisher's Linear Discriminant Analysis (LDA)[3] is done as a nested optimization problem. In the inner optimization problem, Fisher's linear discriminant finds a good split for the given two distinct groups of classes. In the outer optimization problem, *Exchange Method* [13] is used to divide K classes into two groups.

Assuming the data is normally distributed, one dimensional LDA reduces to a second order equation

$$ax^2 + bx + c = 0 \tag{2.1}$$

```
LDT(Node n, Instances x, Features f, Classes c)
    for each feature i in f
        if f_i discrete
            Calculate information gain g_i
            if g_i < bestgain
                bestgain = g_i
                bestfeature = i
            endif
        endif
        if f_i continuous
            Find best split point using LDA
            and exchange method
            Calculate information gain g_i
            if g_i < bestgain
                bestgain = g_i
                bestfeature = i
            endif
        endif
    endfor
    CREATE_CHILDREN(n, x, bestfeature)
    Call LDT for each child node
```

Figure 2.3. LDT Algorithm

and the two candidate best split points are the roots of that equation, where

$$\begin{aligned} a &= s_L^2 - s_R^2 \\ b &= 2(m_L s_R^2 - m_R s_L^2) \\ c &= (m_R s_L)^2 - (m_L s_R)^2 + 2 s_L^2 s_R^2 \log \frac{n_L s_R}{n_R s_L} \end{aligned} \tag{2.2}$$

m_L, m_R are the means and s_L, s_R are the standard deviations of the feature i of the left and right class groups. n_L and n_R are the number of data that are assigned to the left and right class groups respectively.

If the two groups have the same variance, there is only one root. If the variances are different, there are two roots and the one which is between the two means is used. If neither of the two roots is between the means nor there are no roots of the quadratic equation, the middle point of two means is chosen as the split point.

Since we need one pass over the data to find the mean and variance of two groups, complexity of the LDT algorithm is $\mathcal{O}(N)$ for inner optimization. If we make l passes in the outer optimization problem, the complexity of finding best split for a single feature will be $\mathcal{O}(lN)$.

3. Parallel Formulations

3.1. Feature Based Parallelization

Since for each feature, we do the same operations to find the best split for that feature, we can easily parallelize the job at each node by distributing the features to the slave processors. Figure 3.1 gives the pseudocode of this idea. After sending the features and the data that they will process, to slave processors, we receive the best splits and the best information gains for those features. Comparing the best information gains in the host processor, we can easily find the best split and the best feature.

```
FParallel(Node n, Instances x, Features f)
    for each feature i in f
        Submit f_i and n to slave processor
    endfor
    for each feature i in f
        Receive best split and gain
        of feature f_i from slave processor
    endfor
    Find bestsplit and bestfeature
    CREATE_CHILDREN(n, x, bestfeature)
    Call FParallel for each child node
```

Figure 3.1. Feature Based Parallelization

The advantage of feature based parallelization is its simple implementation. Since we have the serial codes of LDT and C4.5 for each feature, we can easily plug that code into the slave processors to find best split for one feature.

According to the dataset we have, the performance of feature based parallelization may change. For example if we have a dataset that contains only discrete features, both LDT and C4.5 reduce to the same algorithm. Since finding the information gain of a discrete feature has a time complexity of $\mathcal{O}(N)$, the load balance will be good.

If there are two classes in the dataset, we do not need a method to divide classes into two groups. So $l = 1$ (number of iterations of exchange method) and LDT will have a time complexity of $\mathcal{O}(N)$ for that feature. In that case, processing discrete and continuous features will have the same complexity and slave processors will have same load. In the case of continuous features, C4.5 will have the same load at each continuous feature but has a significantly large load over discrete features ($\mathcal{O}(NlogN)$ instead of $\mathcal{O}(N)$). If there are more than two classes in the dataset, the time complexity of finding best split for a continuous feature will be $\mathcal{O}(lN)$. Since dividing classes into two class groups may differ from one feature to another feature, the load of slave processors will be imbalanced.

3.2. Node Based Parallelization

As explained above, decision tree construction is done recursively at each decision node. So why not distribute decision node's to processors? Figure 3.2 shows the pseudocode for node parallelization. We use a queue for handling current unexpanded nodes. If there are nodes in the queue, we dequeue them from the queue and send to the slave processor(s) to find the best split for that node. Since LDT and C4.5 algorithms are defined for a single node, each slave processor can call those serial codes to find the best split for that node. After expanding the node(s), produced child nodes are put into the queue to be processed later. Algorithm terminates, when there are no nodes in the queue to be expanded.

Node based parallelization requires the smallest time for communication between host and slave processors. For a single node there is only one message passing, in which host processor sends the instances for that node. Like the feature based parallelization, its implementation is simple, only queue processing should be handled.

If the decision tree is small, processing time of the root node takes a significant amount of time, compared to the whole tree generation. In that case, since we only have one processor job in the starting phase all but one processors will wait the single processor to complete its process. Second disadvantage of the node based parallelization

```
NParallel(Instances x, Features f)
    Queue q = Emptyqueue
    Enqueue(q, RootNode)
    while (Not Empty(q))
        Node = Dequeue(q)
        Submit Node with its instances to slave processor
        Receive best split and bestfeature
        of Node from slave processor
        CREATE_CHILDREN(Node, Node.instances, bestfeature)
        Enqueue(q, Node.child nodes)
    endwhile
```

Figure 3.2. Node Based Parallelization

is the complexity of finding the best split significantly drops in the deeper nodes, because small instances sets come to those nodes. In such cases, the load imbalance may occur between slave processors.

3.3. Data Based Parallelization

We can also divide data into K parts, where we have K processors. At each step of the decision tree algorithm, we send current data to the processors. The slave processors turn back with appropriate statistics. Since the continuous feature phase of LDT and C4.5 algorithm differs, we need different parallelizations for two algorithms. The pseuodocodes of these two parallelizations are given in Figures 3.3 and 3.5.

Figure 3.4 shows a sample execution of data parallel version of C4.5 using three slave processors. There are total 9 instances in the current node and each slave processor has 3 of them. The instances in the slave processors are sorted in $\mathcal{O}(\frac{N}{k}log(\frac{N}{k}))$ time using Quicksort. After sorting, each slave processor sets its iterator to show the first instance. At each step, slave processors send the values their iterators point to, to the host processor, which selects the minimum of them as the split point. Host processor sends the split point to slave processors. After getting the split point, slave processors send frequency statistics of this split point to the host processor. By frequency statistics, we mean total number of elements, those have a feature value smaller or equal to

```
DParallelC45(Instances x, Feature f_i)
    Send x and f_i to each slave processor
    Sort x in each slave processor
    Receive possible split points from slave processor(s)
    Determine the minimum of them
    Send the minimum to the slave processor(s)
    Receive the frequency statistics from slave processor(s)
    Slave processor(s) update their iterator(s) if necessary
    Calculate information gain g_i for that split point
    from the gathered frequency statistics
    Compare g_i with bestgain and update if former is better
    Goto above to receive again possible split points
```

Figure 3.3. Data Based Parallelization of C45

the split point, for each class. By using gathered frequency statistics, host processor can now determine the goodness of the split. Before continuing with the next step, each slave processor updates such that its iterator points to the next feature value.

Figure 3.6 shows a sample execution of parallel version of LDT using 3 slave processors. One important point is that we do not have to sort instances in slave processors in LDT. In step I, each slave processor sends the sum of its feature values of each class to the host processor. In step II, using these sums, host processor easily finds the means of the left and right class groups. The means are sent to the slave processors. In step III, using group means, slave processors find the sum of squares of the differences between the feature values and means. The sums are sent to host processor. In step IV, using the sums, host processor first calculate the standard deviations, coefficients of the second order equation, and roots of that equation. With the roots of the quation, LDT finds the best split point and sends it to the slave processors. In step V, slave processors calculate the frequency statistics for the best split point and send them to host processor. In step VI, host processor finds the goodness of the split using gathered frequency statistics.

The advantage of data based parallelization is its potential for scalability. The datasets in Data Mining usually have large number of instances. Distributing these instances to slave processor(s) equally, makes data base parallelization scalable. Feature

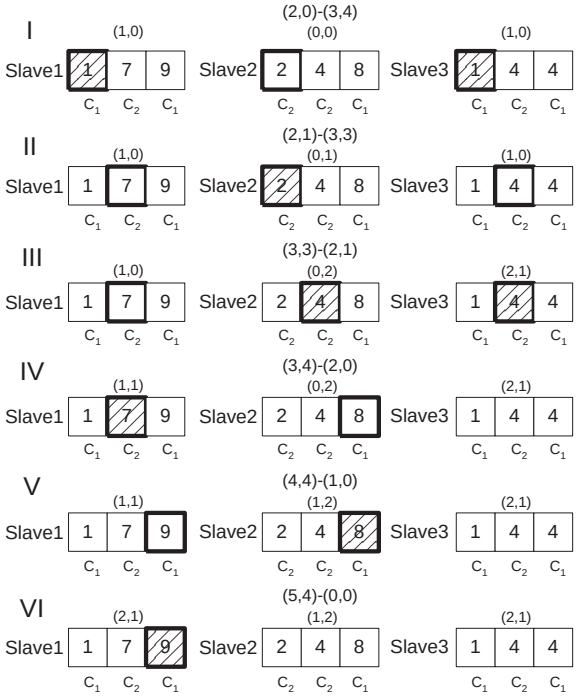

Figure 3.4. Sample execution of Data Parallelization of C45

based parallelization needs large number of features in a dataset to scale well, which does not occur frequently in Data Mining Applications. Node based parallelization needs a large tree to scale well.

The communication cost of data parallelization is much higher compared to both feature based parallelization and node based parallelization.

Second disadvantage of data based parallelization is load imbalance between slave processor(s). Even though each processor started with the same number of training instances at the root node, in deeper nodes the number of training instances belonging to the nodes can vary substantially among processors. For example, processor 1 might have all training instances of a node A, whereas none of node B; processor 2 might have all training instances of the node B, whereas none of node A. When A is selected to expand, processor 1 will do all the job and processor 2 will do nothing and similarly when B is selected to expand, processor 2 will do all the job and processor 1 will do nothing.

DParallelLDT(Instances x, Feature f_i)
Send x and f_i to each slave processor
for each possible class partition C_L, C_R
Send class partition to the slave processor(s)
Receive frequency statistics of that partition
Determine the performance of the partition
endfor
Find the best split point using best class partition
Send the split point to slave processor(s)
Receive frequency statistics from slave processor(s)
Find information gain of the feature f_i using
gathered frequency statistics

Figure 3.5. Data Based Parallelization of LDT

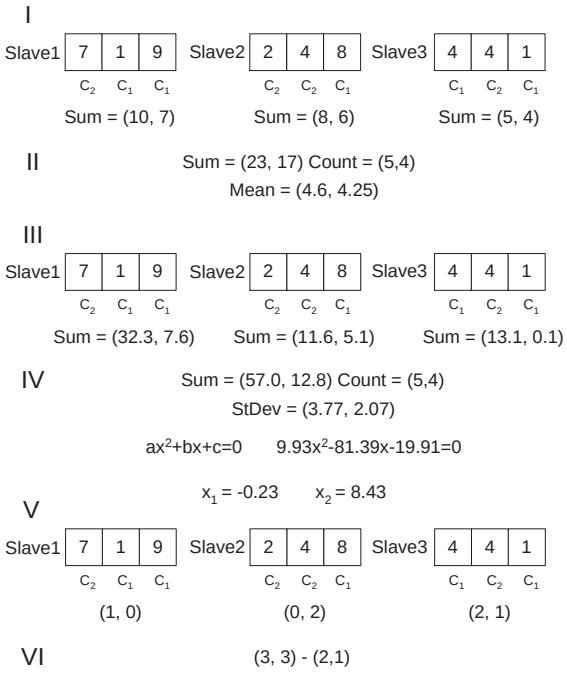

Figure 3.6. Sample execution of Data Parallelization of LDT

4. Experimental Details

4.1. Specifications

In our experiments we use a Beowulf cluster with 24 processors[4]. Each node of the cluster has a Pentium II 400 Mhz processor with 512 KB Cache, 128 MB RAM and an Intel EtherExpress Pro/100+ Network Adapter. All the nodes are connected on HP Procurve 4000M Fast Ethernet Switch. We have Linux 2.2.12 Kernel and Redhat 6.1 distribution on each node. The parallel programs are coded using MPI parallel programming library[7] and compiled using GNU C compiler.

4.2. Datasets

Since the datasets in UCI Machine Learning Repository are small either in sample size or feature size, we get three different datasets from different resources. Attributes of the datasets are given in Table 4.1. To test all three types of parallelizations, for feature based, we get a dataset with a large number of attributes (*Face*), for node based, we get a dataset with a large tree size (*Aibo*), for data based we get datasets with large number of data (*Aibo*, *Census*). Since LDT behaves differently according to the number of classes, we take one dataset with two classes (*Census*) and two others with large number of classes (*Aibo*, *Face*). The performance difference between continuous and discrete features will be tested on the (*Census*) dataset, which has continuous and discrete features.

Table 4.1. Descriptions of the data sets

Set	Class	Size	Discrete	Continuous	Missing
Aibocolor[1]	8	250000	0	3	No
Census[2]	2	100000	33	6	Yes
Face[8]	40	400	0	4096	No

5. Results

In our experiments, we want to check the speedup of the three types of parallelizations of LDT and C4.5. LDT and C4.5 algorithms are parallelized as node based, feature based and data based manner. For each of these six types of parallelizations, we did runs using 1 master processor and a range of slave processors from 1 to 7. We record the time to create a decision tree. Since one run will not be appropriate, we made 10 fold cross-validations for each experiement and took the average time. For feature based parallelization, since *Aibo* dataset has only 3 features, we used maximum 3 slave processors for that dataset.

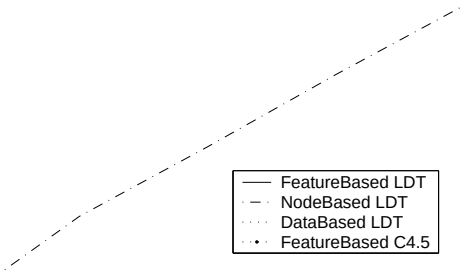

Figure 5.1. Different Parallelization Results on *Aibocolor* Dataset

Figure 5.1 shows the speedup curves of different parallelizations on *Aibocolor* dataset. Since *Aibocolor* dataset has 3 features, Feature Based parallelizations can be done at max for three slave processors. Here C4.5 has larger speedup than LDT. Since *Aibocolor* dataset has more than 2 classes, at different features LDT may work differently (l values may differ because of Exchange Method). On the other hand, C4.5 will behave similarly at each feature. NodeBased Parallelization of LDT has the largest

speedup on this dataset.

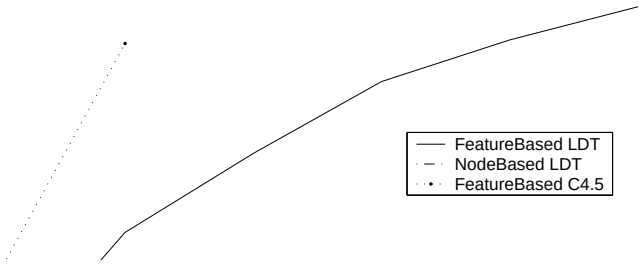

Figure 5.2. Different Parallelization Results on *Face* Dataset

Figure 5.2 shows the speedup curves of different parallelizations on *Face* dataset. As expected, since *Face* dataset has many features, Feature Based parallelizations have better speedups than NodeBased parallelizations. As explained in the above paragraph, C4.5 has better Feature Parallelization than LDT until 3 slave processors.

Figure 5.3 shows the speedup curves of different parallelizations on *Census* dataset. Since this dataset has small number of features, Node based parallelization of LDT has better speedup than Feature Based parallelization.

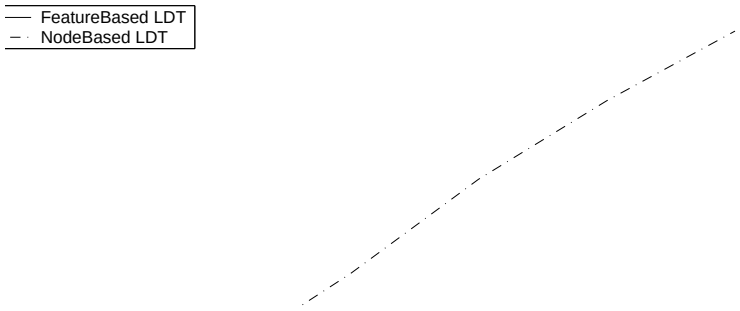

Figure 5.3. Different Parallelization Results on *Census* Dataset

REFERENCES

1. Akın, H. L., Pavlova, P., Yıldız, O. T.: Cerberus 2002, Robocup 2002: Robot Soccer World Cup VI, The 2002 International Robocup Symposium Pre-Proceedings, June 24-25, 2002, Fukuoka, pp.448.

2. UCI Knowledge Discovery in Databases Archive, Census-Income Database: http://kdd.ics.uci.edu/databases/census-income/census-income.html

3. Duda, R. O., Hart, P. E.: Pattern classification and scene analysis. New York: Wiley-Interscience Publication. (1973)

4. Gürdağ, B., Özturan, C., Çağlayan, M. U.: Ağ temelli, ekonomik çok işlemcili bilgisayar geliştirilmesi, Bilişim'99, (1999) (in Turkish).

5. Kufrin, R.: Decision trees on parallel processors. Parallel Processing for Artificial Intelligence 3. (1997) J. Geller, H. Kitano, and C. B. Suttner (Ed.) Elsevier Science.

6. Mehta, M., Agrawal, R., Rissaneh, J.: SLIQ: A fast scalable classifier for data mining. (1996) Proc. of the Fifth International Conference on Extending Database Technology. Avignon. France.

7. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, May 1994.

8. ORL Face Database: http://www.uk.research.att.com/facedatabase.html

9. Quinlan, J. R.: Induction of decision trees. Machine Learning. **1** (1986) 81–106

10. Quinlan, J. R.: C4.5: Programs for Machine Learning (1993) San Mateo, CA:Morgan Kaufmann.

11. Shafer, J., Agrawal, R., Mehta, M.: SPRINT: A scalable parallel classifier for data

mining. Proc. of the 22nd VLDB Conference (1996)

12. Srivastava, A., Han, E., Kumar, V., Singh, V.: Parallel Formulations of Decision-Tree Classification Algorithms Data Mining and Knowledge Discovery. **3** (1999) 237–261

13. Yıldız, O. T., Alpaydın, E.: Linear Discriminant Trees, 17th International Conference on Machine Learning, P. Langley (Ed.), pp. 1175–1182, Morgan Kaufmann. (2000)