Efficient Parallel Classification Using Dimensional Aggregates

Sanjay Goil Alok Choudhary*
Department of Electrical & Computer Engineering
Northwestern University,
Technological Institute,
2145 Sheridan Road, Evanston, IL-60208
{sgoil,choudhar}@ece.nwu.edu

Abstract

Multidimensional aggregates are frequently computed to improve query performance in Online Analytical Processing applications. We present a new method for decision tree based classification trees using the aggregates computed in the multidimensional data model. The structure imposed on data in a explicit multidimensional storage mechanism leads to efficient dimensional operations. Decision tree based classification algorithms perform computations to find the best split point at each node of the tree. Efficient computation of the split in the decision tree can be done by using the one-dimensional aggregates if the cell values are the class-id values, and counts are maintained for each class. This is used repeatedly at the nodes of the decision tree to calculate splits and manage data. Previous parallel approaches for decision-tree based classification use sorted attribute lists and hash tables to compute the split point and split the data appropriately. The amount of data communicated is proportional to the product of number of records in the training set, and the number of dimensions, at each level of the tree, in the worst case. Parallel formulation of our approach uses data communication proportional to the product of the sum of cardinality of all dimensions and the number of non-classified nodes at each level of the tree. Communication volume is greatly reduced in our approach and is done in one phase of communication at each level of the tree, by coalescing messages. Preliminary results from our experiments on a coarse-grained, distributed memory parallel machine (IBM-SP2) show good performance.

*This work was supported in part by NSF Young Investigator Award CCR-9357840 and NSF CCR-9509143.

1 Introduction

Classification of large data sets has received considerable attention in the data mining literature recently. The objective of classification is to build a model of the classifying attribute based upon the other attributes of the record. A set of sample records called the training data set is given, consisting of several attributes. Attributes can either be *continuous*, if they come from an ordered domain, or *categorical*, if they are from an unordered domain. One of the attributes is the *classifying* attribute that indicates the *class* to which the record belongs. Several classification models have been used in the past, notably neural networks [1], genetic algorithms [2], and decision trees [3, 4]. Among these models, the decision tree models are considered to be the most useful in the domain of data mining because they are relatively inexpensive to construct, easy to interpret and easy to integrate with data base systems. Also, for a variety of problem domains they yield comparable or better accuracy as compared to the other models [1].

A decision tree recursively partitions the training set until each partition consists entirely or dominantly records from one class. Each *non-leaf* node of the tree contains a split point which is a test on an attribute and determines how the data is partitioned. Once the decision tree is built from the training set it can be used to classify future instances. The decision-tree based classifiers that can handle large data sets are important because use of larger data sets improves the classification accuracy [1].

Previous work in classifying large data sets has been to use sampled data sets or multiple partitions of the data set [1, 5]. Recent work has focused on using the entire data set, in classifiers like SLIQ [6] and SPRINT [7]. A parallel classifier in the same spirit has been developed in ScalParC [8]. Classifiers like CART [3] and C4.5 [4] perform sorting at every node of the decision tree, which makes them expensive for large data sets since disk-based sorting is required at each node. The approach of SPRINT, SLIQ is to sort the continuous attribute once in the beginning and maintain the sorted order in the subsequent splitting steps. Separate lists are kept for each attribute which maintains a record identifier for each sorted value. In the splitting phase the same records need to be assigned to a node, which may be in a different order in the different attribute lists. A hash table is used to provide a mapping between record identifiers and the node to which it belongs after the split. This mapping is then probed to split the attribute lists in a consistent manner. A framework for instantiating several of these algorithms is presented in [9], which uses attribute value and class-label pairs (AVC-sets) to make the splitting criteria decision. It is also a greedy top-down approach as the others, except that it works on the AVC-sets at each node of the decision tree. This allows it to use memory more efficiently and perform much better than the attribute-list approaches.

Table 1: Training set data

Row-id	Age	Car-Color	Gender	Class-id
0	10	Green	F	0
1	50	Blue	M	1
2	40	Yellow	F	0
3	30	Green	F	0
4	20	Red	M	1
5	40	Blue	M	0
6	20	Yellow	M	1

Table 1 is an example training set with three attributes, Age, Car color and Gender, and a class attribute. Age is a continuous attribute, whereas both Car color and Gender are categorical attribute. For a categorical attribute having c distinct classes it is assumed that the splitting decision forms c partitions, one for each of its values. Figure 1(a) shows the classification tree for it. At each node the attribute to split is chosen that best divides the training set. Several splitting criteria have been used in the past to evaluate the goodness of a split. Calculating the *gini* index is commonly used [3]. This involves computing the frequency of records

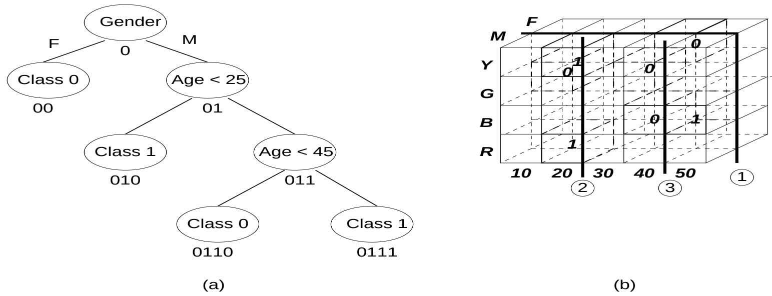


Figure 1: (a) Classification tree for training set (b) Classification tree embedded on a cube

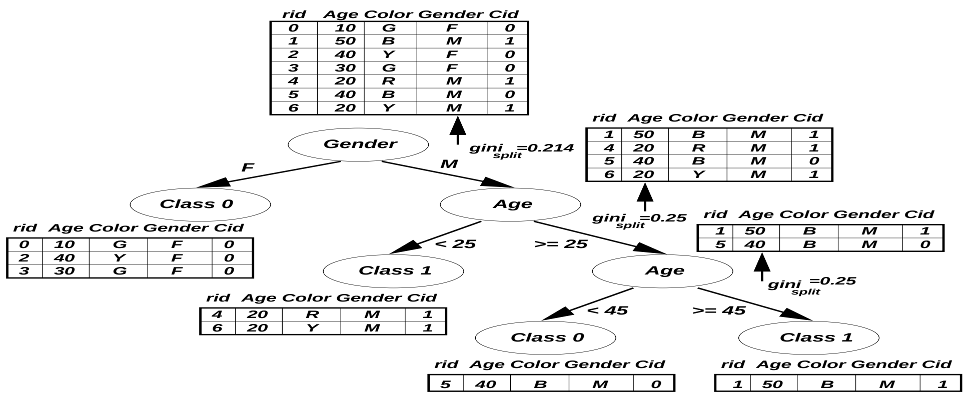


Figure 2: Gini index calculation for the attributes and node splitting

of each class in each of the partitions. If a parent node having n records and c possible classes is split into p partitions, the *gini* index of the i^{th} partition is $gini_i = 1 - \sum_{j=1}^{c}(n_{ij}/n_i)^2$, where n_i is the total number of records in partition i, of which n_{ij} records belong to class j. The *gini* index of the total split is given by $gini_{split} = \sum_{i=1}^{p}(n_i/n)gini_i$. The attribute with the least value of $gini_{split}$ is chosen to split the records at that node. The matrix n_{ij} is called the *count matrix*. The count matrix needs to be calculated for each evaluated split point for a continuous attribute.

Categorical attributes have only one count matrix associated with them, hence computation of the gini index is straightforward. For the continuous attributes an appropriate splitting value has to be determined by calculating the $gini_{split}$ and choosing the one with the minimum value. If the attribute is sorted then a linear search can be made for the optimal split point by evaluating the gini index at each attribute value. The count matrix is calculated at each possible split point to evaluate the $gini_{split}$ value. The *gini* index calculations and the node splits for the example above are given in Figure 2. At Node 0, the attribute *Gender* yields the optimal $gini_{split}$ value of 0.214. This creates a split with one partition with M values for gender and another with F values. After this split is made, two child nodes are created. The record values need to be partitioned consistently between the two nodes for the *split-attribute* and the *non-split attributes*. Splitting the split-attribute is straightforward by adjusting pointer values. The challenge is to split the non-split attributes efficiently. Existing implementations such as SPRINT and ScalParC maintain a mapping of the row-id and class-id with the values assigned to each node. The values are split physically among nodes, such that the continuous attribute maintain their sorted order in each node to facilitate the sequential scan for the next split determination phase. A hash list maintains the mapping of record ids to nodes. The record ids in the lists for non-splitting attributes are searched to get the node information, and perform the correct split.

2 Proposed Classification using Multidimensional Aggregates

Multidimensional analysis, OLAP queries and association rule mining are performed efficiently using the materialized aggregates in the data cube. A multidimensional chunk based infrastructure for OLAP and multidimensional analysis for high dimensional data is developed in [10], which optimizes the building of the data cube operator [11]. In this model an attribute is treated as a dimension, and records are points in a multidimensional space. Dimensional operations can be performed more efficiently in such a model since a structure is imposed in the storage of data. Multidimensional arrays are the most intuitive and simple structures for this. However, data sets with large dimension cardinalities and a high number of dimensions cannot be handled using arrays. Also, most data sets are sparse and multidimensional arrays lead to redundant storage in such a scenario. We have used a chunk based implementation to sparse data in a bit-encoded sparse structure (BESS) which encodes the indices of the element in a chunk. Dimensional operations are efficiently performed on compressed chunks, which allow a large number of dimensions to be used. Details can be found in [12].

We propose that classification trees can be built using structure imposed on data using the multidimensional data model. Gini index calculation relies on the count matrix which can be efficiently calculated using the dimensional model. Each populated cell represents a record in the array. For the base cube (which is a multidimensional representation of the records without any aggregation) the class value of the record is stored in each cell. The gini index calculation uses the count matrix which has information about the number of records in each partition belonging to each possible class.

To evaluate split points for a continuous attribute the $gini_{split}$ needs to be evaluated for each possible split point in a continuous attribute and once for a categorical attribute. This means the aggregate calculations present in each of the 1 dimensional aggregates can be used if they have number of records belonging to each class. Therefore for each aggregate we store the number of records in each class. Figure 3(a) gives an example training set with two dimensions, A, a continuous dimension and B a categorical dimension and two class values 0 and 1.

Figure 3(b) is the corresponding multidimensional model. The continuous dimensions A is stored in the sorted order. The aggregates store the number of records mapping to that cell for both classes 0 and 1. To calculate the $gini_{split}$ for the continuous attribute attribute A it is now easy to look at the A aggregate and sum the values belonging to both classes 0 and 1 on both sides of the split point under consideration to get the count matrix. Gini index calculation is done on an attribute list which in the case of a multidimensional model is a dimension. Count matrix is repeatedly calculated on the sorted attribute list which is readily available in the cube structure as a higher level one dimensional aggregate. Each dimension is sorted in the dimensional structure as shown in Figure 3(b).

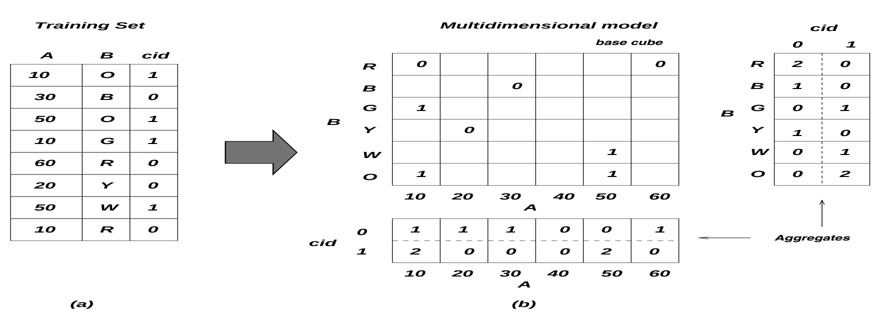


Figure 3: (a) Training set records (b) corresponding multidimensional model

Figure 4 illustrates the classification tree building process using the multidimensional model and the aggregates maintained at the highest level of the cube structure, one for each dimension.

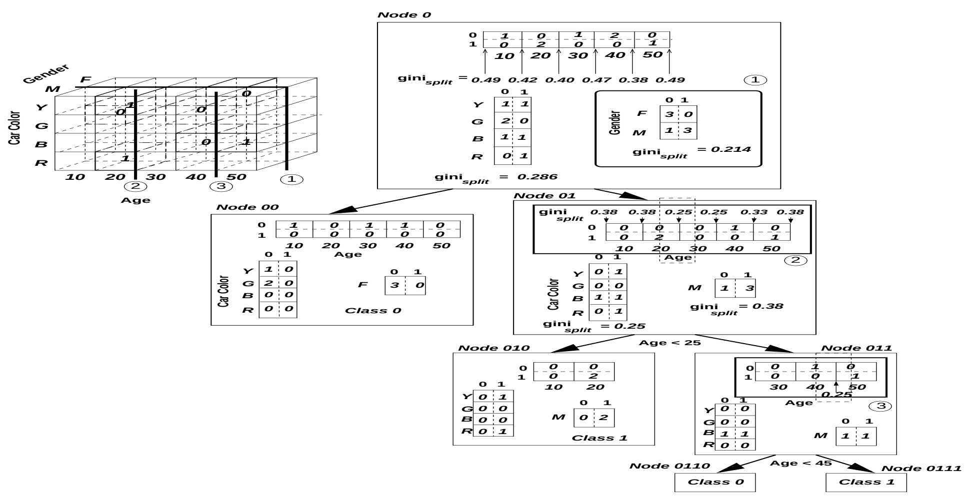


Figure 4: Gini index calculation for the attributes and node splitting with multidimensional aggregates

The challenge is to calculate the one dimensional aggregates efficiently and keep them updated to reflect the partitions after each split. The simple method of computing each level 1 aggregate is to do so from base data. Some alternative strategies that enumerate some intermediate values to optimize the computation of the one-dimensional aggregates have been described in [13].

3 Related Work

In this section we will briefly discuss the previous efforts in parallel classification [14, 7, 8] on distributed memory machines, significant of which are parallel SPRINT [7] and ScalParC [8] since they do not require resorting at each node. Classification on fine-grained shared memory machines has addressed in [15]. The issues involved on such architectures are much different than distributed memory machines and hence we do not discuss that work in this paper.

Parallelization of tree methods, especially multidimensional binary search trees (k-d trees), quadtrees and octrees for hierarchical methods have received considerable attention in the areas of scientific processing [16]. Classification trees fall under a similar paradigm.

The parallel version of SPRINT partitions each attribute list by sorting each one using a parallel sort with probabilistic splitting. This gives approximately equal sections to each processor of each sorted attribute list. Split points are found by first doing a **prefix sum** operation for the count of values below and above the first split point on each processor followed by building the count matrices locally on each processor. After calculating the gini index locally, the processors communicate to determine which split point has the lowest value. Since there is only one count matrix for categorical attributes, they are constructed locally and one processor collects the global count and calculates the gini values.

Each processor splits its splitting attribute list locally. *rids* are collected from all processors to build the hash table on each processor. The non-splitting attribute lists then probe this table with *rids* to determine the split locally. This has a space complexity of $O(N)$ and makes it unscalable in memory requirements. ScalParC on the other hand maintains a distributed hash table for the splitting phase which has a space complexity of $O(N/P)$. However, the splitting phase is slightly different in their case. A distributed hash table, called the *node table* is maintained by hashing a rid with a hash function $h(rid) = (p = rid\ div\ N/P, l = rid\ mod\ N/P)$, where the first field is the processor number and the second is the local index on that processor. After the splitting decision, each processor uses the $rids$ of the split attribute to construct hash buffers with $(l, child)$ entry destined for all processors p calculated by the hash function. An **all-to-all personalized** communication phase exchanges these to update the distributed node table. For each non-split attribute list, the distributed node table is queried by filling out a communication buffer with $rids$ (enquiry buffer) and sending it to the processor p which then fills it with the child label and sends it back. Two **all-to-all personalized** communication phases are needed to achieve this.

The size of the node table at each level is usually of size $O(N/P)$. Also, each processor sends $O(N/P)$ elements from each of the non-splitting attribute list. However, there are cases when a processor has to hash all global $rids$ at some level and the other processors need to send $O(N)$ elements to be queried by the node table on a single processor [8]. Hence, the worst case complexity is $O(N)$.

4 Parallel Classification using Multidimensional Aggregates

Parallel classification on the multidimensional cube is similar to the sequential classification algorithm, except the fact that each processor calculates the aggregates locally and then needs to update counts for each partitioned dimension from other processors. This is done by a communication phase which calculates a **prefix sum** on the first split point for continuous attributes since each split point is evaluated by computing the $gini_{split}$. For categorical attributes, however, a processor can sum the counts for a value of the categorical dimension and calculate the gini index. Note that separate processors can compute the gini value for a categorical attribute.

4.1 Dimension partitioning and gini index calculation

Figure 5 shows a one dimensional partitioning of the multidimensional base cube and the associated local aggregate calculations. Each processor builds the aggregate locally and for each continuous attribute determines the gini index for the values that lie in its partition. For non-distributed dimensions, each processor locally calculates the aggregates and then does a **reduce all (sum)** on the aggregates for each such dimension. Each processor then works on a part of the aggregate list to calculate the gini index values. This results in computation being partitioned between processors. If the size of the dimension i is d_i, then each processor $P_j, 0 \leq j < P$, calculates the gini index from $(d_i/P)P_j$ to $(d_i/P)P_{j+1}$ for each dimension. For categorical attributes also the computation can be distributed similarly if a **reduce all** is performed on the aggregated dimensions. Otherwise a **reduce (sum)** operation gets the aggregates on a single processor which does the calculation. Since dimensions with categorical attributes are usually small we use the latter approach.

Another alternative is to perform a two dimensional partitioning by selecting the two largest dimensions and partitioning the multidimensional base cube on a 2-dimensional processor grid as shown in Figure 5. This may provide better load balancing in some cases. The communication pattern for the two distributed dimension changes but remains the same for the rest. The non-distributed dimensions get the aggregates locally and do a **reduce all (sum)** to sum each dimensional aggregate independently on each processor. Each processor then works at calculating the gini index in contiguous portions as in the one-dimensional partitioning case. For the distributed dimension A in the figure the reduce for the dimension on processor

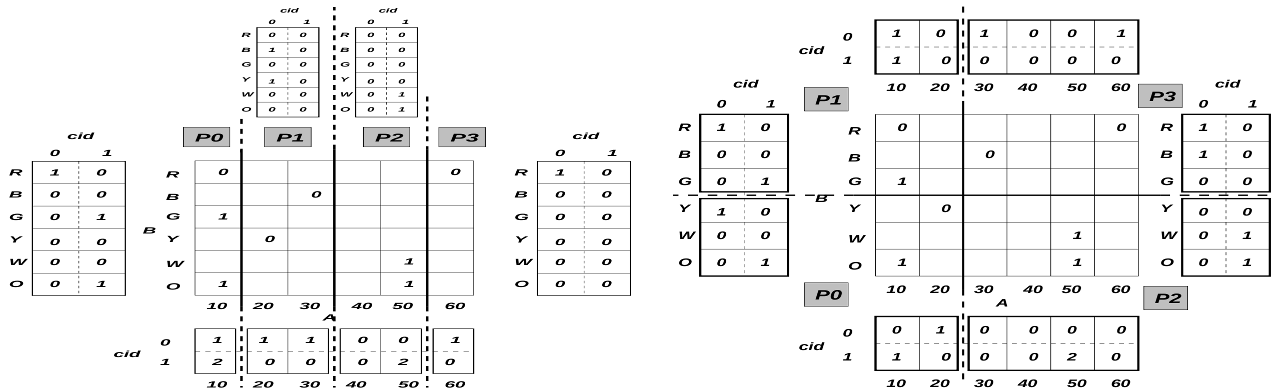


Figure 5: One and two dimensional partitioning of dimensions

P_j is done on processor $(P_j/P_Y)P_Y$, where P the number of processors is divided into a two dimensional grid in the dimensions X and Y as $P_X \times P_Y$. The processors $P_0 + i * P_Y, i = 0, \ldots P_X$ then calculate the local sums of class ids and need a **prefix sum** to get the values updated across processor boundaries with the number above and below in each class. Each of these processors then calculates the gini index for the local dimension values.

Similarly, for dimension B a **reduce** operation is done on processor $P_j \% P_Y$ and again a **prefix sum** is done between processors $P_0 + i, i = 0, \ldots P_Y$ to calculate the gini index locally.

4.2 Node splits

Once the $gini_{split}$ is calculated for all attributes (dimensions), each processor picks out the minimum and the attribute it is related to. A **Reduce (minimum)** gets the minimum and the attribute which is the *split* attribute. Each processor then partitions the split attribute locally. Each processor maintains a mapping of dimension indices to the node they belong to, as they are split. In other words, a global tree representation is present on each node. Each node keeps the dimensional boundaries of the dimensions split, defining a hypercube the node encapsulates. The aggregates are then calculated within the node boundaries. The communication required is for each dimension, but now there are multiple count values, one for each active node which has not yet been classified.

Let n_a be the number of attributes (dimensions) divided into n_d, the set of distributed dimensions and n_u, the set of non-distributed dimensions. For non-distributed dimensions these need to be aggregated using a **Reduce (sum)** on all processors to distribute the computation of the gini index calculation. The amount of data communicated at a level k of the classification tree is $O(\mathcal{N}_n^k |D_u|)$, where $\mathcal{N}_n^k$ is the number of active nodes at a level k, and $|D_u|$ is the sum of dimension sizes of dimensions $d_i \in n_u$. For a distributed dimension a local aggregation is done for each dimension for each node and then a **Parallel prefix (sum)** is done. The data communicated for this operation at level k is $O(\mathcal{N}_n^k n_d)$. The total data communicated at level k of the classification tree is then $O(\mathcal{N}_n^k(|D_u| + n_d)) = O(\mathcal{N}_n^k |D_a|)$, $|D_a| = |D_u| + |D_d|$, because the number of split dimensions is usually one or two. This is the case for our implementation where the optimization of message-coalescing, reduces the number of communication messages by combining the messages as one buffer.

For the scalable record based classification algorithm (ScalParC) the complexity of communication at each level is $O(n_a \frac{N}{P})$ at each level and $O(n_a N)$ in the worst case, where N is the size of the training set. The number of active nodes at any level is much smaller than the size of the training set. Also, the sum of dimension sizes, $|D_a|$, is also much smaller when compared to N. Thus, $\mathcal{N}_n^k \times |D_a| < N$, for a large N. This makes the overall communication requirement of our multidimensional classification algorithm better. Computation of the 1-D aggregates and the gini index calculations has the time complexity $O(\frac{N}{P} + \mathcal{N}_n D_a)$ in our method, whereas ScalParc requires $O(\frac{N}{P} n_a)$, to compute the gini indices and $O(N n_a)$ in the worst case.

A **broadcast** of the prefix sum from the last processor is followed by the local calculation of the gini index and a **Reduce (minimum)** for the gini value which determines the attribute used for the split. Notice that for a non-distributed categorical attribute B the calculation is distributed across processors by letting each processor work on a section since all information is available on each processor as a result of the reduce. Suppose the $gini_{split}$ results in selecting $A = 25$ as the split point in the example.

Figure 6 shows the split at each processor into L and R nodes labeled 00 and 01 respectively. The counts of each dimension are done for each node on each processor. The steps for parallel gini index calculations are done for each node on every processor. Communication can be concatenated for the nodes at each level following the idea of concatenated parallelism in [16] for each active node.

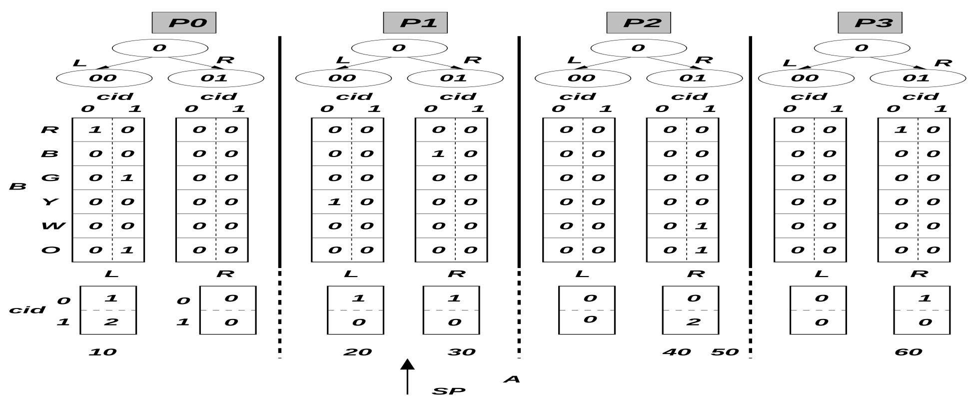


Figure 6: Node split of dimension $A = 25$, and the count arrays for each node

4.3 Communication Optimizations

Only one collective communication operation is performed for all nodes at each level. This is due to the fact that information that needs to be communicated for all nodes is stored contiguously. Each node has the counts of class id. values for each dimension. This is combined using a **Reduce (sum)** operation across all nodes as described in a previous section. The count matrices for each dimension are allocated contiguously for each active node as shown in Figure 7. Each node allocates memory for each dimension, d_i, for each class. At each level only the unclassified nodes (active nodes) are represented and need to participate. Algorithm 1 describes the steps of the overall parallel algorithm for classification.

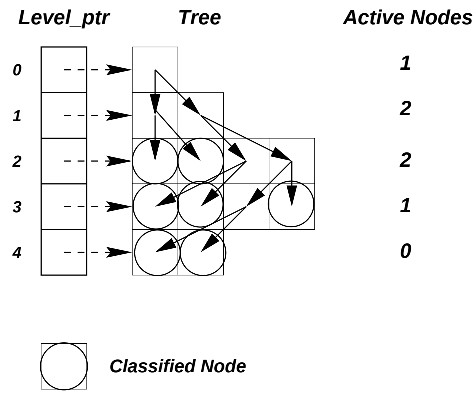


Figure 7: Tree structure at each processor, nodes being stored contiguously to coalesce messages for communication at each level

Algorithm 1 *Multidimensional Classification Algorithm*

/* set the root node to represent the entire multidimensional space */
for $i \leftarrow 0$ to k
    $begin_i = 0$
    $end_i = |d_i|$
$n\_active\_nodes = 1$;
while($n\_active\_nodes > 0$)
    For each chunk c
        Get chunk boundary values in each dimension
        For each active node $j, (0 \leq j < n\_active\_nodes)$
            Determine if chunk c is contained within node j,$[begin_i, end_i)$, for all i, else if it overlaps or else not in range.
    For each active node $j, (0 \leq j < n\_active\_nodes)$
        Compute the contribution of chunk c to 1-dimensional aggregate if the chunk is inclusive or overlapping to the node j.
    One **Reduce (sum)** communication operation for consolidating k 1-dimensional aggregates on all processors.
    For each active node $j, (0 \leq j < n\_active\_nodes)$
        Check if node j is already classified within the threshold level.
        If the node is *not* classified
            For each attribute $i, 0 \leq i < k$,
                Compute $gini$ indices for each dimension i recording the minimum value and corresponding i ($split_i$) and the split point.
            Update for the child node, $begin_i$ and end_i for $split_i$ using the split point and copy the others from the parent node j.
            Add the number of child nodes to the count of active nodes.
    Update n_active_nodes as the count of child nodes.
end

---

5 Preliminary performance Results

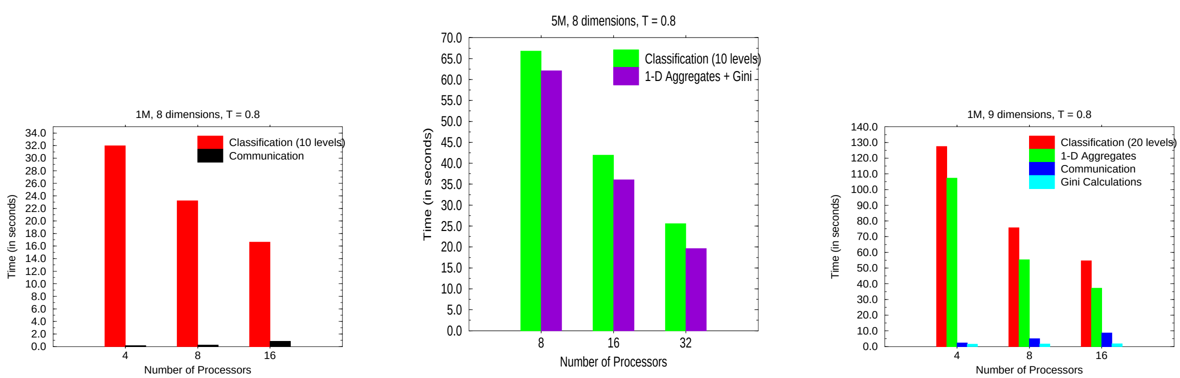
We use the synthetic data generator introduced by Agrawal et al. in [17]. It is a widely used synthetic set used by many others, primarily because there are no large real data sets available.

The synthetic set has nine attributes as shown in Table 2. $\mathcal{U}(x : y)$ denotes the integer uniform distribution with values $v : x \leq v \leq y$. The data set can be generated with different classification functions that assign labels to the records produced. We use Function 1, which is the default, for our performance study. We have used various subsets of the data to evaluate the performance of our algorithm, varying the number of dimensions, the number of records and the classification threshold. We observe good speedup and scale-up performance for our experiments performed mainly on a 16 node IBM SP2 with thin nodes (120Mhz processor) running AIX and having 128 MB main memory per processor. We present a subset of our preliminary results in this section to illustrate performance. More results will be given in the final version of the paper.

Table 2: The sizes of the dimensions in data set used

Predictor Attribute	Distribution	Max. number of entries
Salary	$\mathcal{U}(20K, 150K)$	131
Commission	0, if(salary > 75K) else $\mathcal{U}(10K, 75K)$	66
Age	$\mathcal{U}(20, 80)$	61
Education	$\mathcal{U}(0, 4)$	5
Car	$\mathcal{U}(1, 20)$	20
Zip Code	$\mathcal{U}$(nine zip codes)	9
House Value	$\mathcal{U}(0.5 \times k \times 100K, 1.5 \times k \times 100K)$ k depends on ZipCode	1351
Home Years	$\mathcal{U}(0, 30)$	31
Loan	$\mathcal{U}(0, 500K)$	501

Figure 8(a),(b) shows the time for classifying 1 million and 5 million records with 8 dimension (again without Loan attribute) and classification threshold value set to T = 0.8. The 5 million records data set is We observe that most of the time is taken for the 1 dimensional aggregate calculations and the gini computations, a small fraction taken by communication and maintenance of the classification tree.



(a) (b) (c)

Figure 8: Classification of a 8 dimensional data set for 10 levels, with classification threshold T = 0.8 for (a) 1M records (b) 5M records (c) Classification of a 9 dimensional data set for 20 levels, with classification threshold T = 0.8 for 1M records

Figure 8(c) shows the classification performance for a 9 dimensional data set with 1 million records. The major component of the computation is the 1-dimensional aggregate calculations for all nodes. This is the parallel component and we see good speedup as the number of processors is increased. The other phases of communication and gini index calculations as shown in the figure are small portions of the entire time. Communication increases slightly as number of processors increase, due to the increase in p term in the complexity of the **Reduce** global communication operation. We are currently running experiments with larger data sets and will include the results in the final version.

6 Conclusion

Classification using decision trees is performed on an explicit multi-dimensional storage scheme. The split point for a node in the classification tree is chosen after computing the counts of records belonging to each class for each attribute value, in each dimension. A multidimensional representation make a 1-dimensional computation for each dimension from the base cube very efficient. A parallel framework is used to parallelize this calculation, and one round of communication is required at each level of the tree by concatenating (coalescing) communication for each node together. Results on a synthetic benchmark [17], widely used for classification algorithms performance, shows good parallel performance.

References

[1] D. Michie, D.J. Spiegelhater, and C.C. Taylor. *Machine Learning, Neural and Statistical Classification.* Ellis Horwood, 1994.

[2] D.E Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Morgan Kaufmann, 1989.

[3] L. Breiman, J.H Friedman, R.A Olshen, and C.J Stone. *Classification and Regression Trees.* Wadsworth, Belmont, 1984.

[4] J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993.

[5] P.K Chan and S.J Stolfo. Meta-learning for multistrategy and parallel learning. In *Proc. International Workshop on Multistrategy Learning*, 1993.

[6] M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A fast scalable classifier for data mining. In *Proc. of the Fifth International Conference on Extending Database Technology*, March 1996.

[7] J.C. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proc. 22th International Conference on Very Large Databases*, September 1996.

[8] M. Joshi, G. Karypis, and V. Kumar. ScalParC: A new scalable and efficient parallel classification algorithm for mining large datasets. In *Proc. International Parallel Processing Symposium*, March 1998.

[9] J. Gehrke, R. Ramakrishnan, and V. Ganti. RainForest - A Framework for Fast Decision Tree Construction of Large Data Sets. In *Proc. 24th International Conference on Very Large Databases*, 1998.

[10] S. Goil and A. Choudhary. High performance multidimensional analysis and data mining. In *Proc. SC98: High Performance Networking and Computing Conference*, November 1998.

[11] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proc. 12th International Conference on Data Engineering*, 1996.

[12] S. Goil and A. Choudhary. Design and Implementation of a Scalable System for Multidimensional Analysis and OLAP. In *Proc. International Parallel Processing Symposium*, April 1999.

[13] S. Goil and A. Choudhary. Parallel classification using the multidimensional data model (under preparation). Technical Report CPDC-9904-006, Northwestern University, January 1999.

[14] D.J Fifield. *Distributed Tree construction from large data sets.* Bachelor's Honors Thesis, Australian National University, 1992.

[15] M. Zaki, C.T. Ho, and R. Agrawal. Scalable parallel classification for data mining on shared-memory multiprocessors. In *Proc. International Conference on Data Engineering*, March 1999.

[16] I. Al-furaih, S. Aluru, S. Goil, and S. Ranka. Parallel construction of multidimensional binary search trees. In *Proc. International Conference on Supercomputing*, May 1996.

[17] R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, December 1993.

[18] E. Keogh C. Blake and C.J. Merz. *UCI Repository of machine learning databases.* University of California, Irvine, http://www.ics.uci.edu/~mlearn/MLRepository.html, 1998.