SPRINT: A Scalable Parallel Classifier for Data Mining

John Shafer* Rakesh Agrawal Manish Mehta

IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120

Abstract

Classification is an important data mining problem. Although classification is a well-studied problem, most of the current classification algorithms require that all or a portion of the the entire dataset remain permanently in memory. This limits their suitability for mining over large databases. We present a new decision-tree-based classification algorithm, called SPRINT that removes all of the memory restrictions, and is fast and scalable. The algorithm has also been designed to be easily parallelized, allowing many processors to work together to build a single consistent model. This parallelization, also presented here, exhibits excellent scalability as well. The combination of these characteristics makes the proposed algorithm an ideal tool for data mining.

1 Introduction

Classification has been identified as an important problem in the emerging field of data mining[2]. While classification is a well-studied problem (see [24] [16] for excellent overviews), only recently has there been focus on algorithms that can handle large databases. The intuition is that by classifying larger datasets, we will be able to improve the accuracy of the classification model. This hypothesis has been studied and confirmed in [4], [5], and [6].

In classification, we are given a set of example records, called a *training set*, where each record consists of several fields or *attributes*. Attributes are either *continuous*, coming from an ordered domain, or *categorical*, coming from an unordered domain. One of the attributes, called the *classifying* attribute, indicates the *class* to which each example belongs. The objective of classification is to build a model of the classifying attribute based upon the other attributes. Figure 1(a) shows a sample training set where each record represents a car-insurance applicant. Here we are interested in building a model of what makes an applicant a high or low insurance risk. Once a model is built, it can be used to determine the class of future unclassified records. Applications of classification arise in diverse fields, such as retail target marketing, customer retention, fraud detection and medical diagnosis[16].

Several classification models have been proposed over the years, e.g. neural networks [14], statistical models like linear/quadratic discriminants [13], decision trees [3][20] and genetic models[11]. Among these models, decision trees are particularly suited for data mining [2][15]. Decision trees can be constructed relatively fast compared to other methods. Another advantage is that decision tree models are simple and easy to understand [20]. Moreover, trees can be easily converted into SQL statements that can be used to access databases efficiently [1]. Finally, decision tree classifiers obtain similar and sometimes better accuracy when compared with other classification methods [16]. We have therefore focused on building a scalable and parallelizable decision-tree classifier.

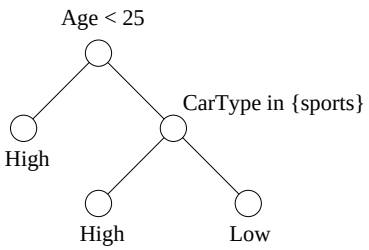
A decision tree is a class discriminator that recursively partitions the training set until each partition consists entirely or dominantly of examples from one class. Each non-leaf node of the tree contains a *split point* which is a test on one or more attributes and

*Also, Department of Computer Science, University of Wisconsin, Madison.

Proceedings of the 22nd VLDB Conference Mumbai(Bombay), India, 1996

rid	Age	Car Type	Risk
0	23	family	High
1	17	sports	High
2	43	sports	High
3	68	family	Low
4	32	truck	Low
5	20	family	High

(a) Training Set



(b) Decision Tree

Figure 1: Car Insurance Example

determines how the data is partitioned. Figure 1(b) shows a sample decision-tree classifier based on the training set shown in Figure 1a. ($Age < 25$) and ($CarType \in \{sports\}$) are two split points that partition the records into High and Low risk classes. The decision tree can be used to screen future insurance applicants by classifying them into the *High* or *Low* risk categories.

Random sampling is often used to handle large datasets when building a classifier. Previous work on building tree-classifiers from large datasets includes Catlett's study of two methods [4][25] for improving the time taken to develop a classifier. The first method used data sampling at each node of the decision tree, and the second discretized continuous attributes. However, Catlett only considered datasets that could fit in memory; the largest training data had only 32,000 examples. Chan and Stolfo [5] [6] considered partitioning the data into subsets that fit in memory and then developing a classifier on each subset in parallel. The output of multiple classifiers is combined using various algorithms to reach the final classification. Their studies showed that although this approach reduces running time significantly, the multiple classifiers did not achieve the accuracy of a single classifier built using all the data. Incremental learning methods, where the data is classified in batches, have also been studied [18][25]. However, the cumulative cost of classifying data incrementally can sometimes exceed the cost of classifying the entire training set once. In [1], a classifier built with database considerations, the size of the training set was overlooked. Instead, the focus was on building a classifier that could use database indices to improve the retrieval efficiency while classifying test data.

Work by Fifield in [9] examined parallelizing the decision-tree classifier ID3 [19] serial classifier. Like ID3, this work assumes that the entire dataset can fit in real memory and does not address issues such as disk I/O. The algorithms presented there also require processor communication to evaluate any given split point, limiting the number of possible partitioning schemes the algorithms can efficiently consider for each leaf. The Darwin toolkit from Thinking Machines also contained a parallel implementation of the decision-tree classifier CART [3]; however, details of this parallelization are not available in published literature.

The recently proposed SLIQ classification algorithm [15] addressed several issues in building a fast scalable classifier. SLIQ gracefully handles disk-resident data that is too large to fit in memory. It does not use small memory-sized datasets obtained via sampling or partitioning, but builds a single decision tree using the *entire* training set. However, SLIQ does require that some data per record stay memory-resident all the time. Since the size of this in-memory data structure grows in direct proportion to the number of input records, this limits the amount of data that can be classified by SLIQ.

We present in this paper a decision-tree-based classification algorithm, called SPRINT[1], that removes all of the memory restrictions, and is fast and scalable. The algorithm has also been designed to be easily parallelized. Measurements of this parallel implementation on a shared-nothing IBM POWERparallel System SP2 [12], also presented here, show that SPRINT has excellent scaleup, speedup and sizeup properties. The combination of these characteristics makes SPRINT an ideal tool for data mining.

The rest of the paper is organized as follows: In Section 2 we discuss issues in building decision trees and present the serial SPRINT algorithm. Section 3 describes the parallelization of SPRINT as well as two approaches to parallelizing SLIQ. In Section 4, we give a performance evaluation of the serial and parallel algorithms using measurements from their implementation on SP2. We conclude with a summary in Section 5. An expanded version of this paper is available in [22].

2 Serial Algorithm

A decision tree classifier is built in two phases [3] [20]: a growth phase and a prune phase. In the growth phase, the tree is built by recursively partitioning the

[1] SPRINT stands for Scalable PaRallelizable INndution of decision Trees.

data until each partition is either "pure" (all members belong to the same class) or sufficiently small (a parameter set by the user). This process is shown in Figure 2. The form of the split used to partition the data depends on the type of the attribute used in the split. Splits for a continuous attribute A are of the form $value(A) < x$ where x is a value in the domain of A. Splits for a categorical attribute A are of the form $value(A) \in X$ where $X \subset domain(A)$. We consider only binary splits because they usually lead to more accurate trees; however, our techniques can be extended to handle multi-way splits. Once the tree has been fully grown, it is pruned in the second phase to generalize the tree by removing dependence on statistical noise or variation that may be particular only to the training set.

The tree growth phase is computationally much more expensive than pruning, since the data is scanned multiple times in this part of the computation. Pruning requires access only to the fully grown decision-tree. Our experience based on our previous work on SLIQ has been that the pruning phase typically takes less than 1% of the total time needed to build a classifier. We therefore focus only on the tree-growth phase. For pruning, we use the algorithm used in SLIQ, which is based on the Minimum Description Length principle[21].

```
Partition(Data S)
    if (all points in S are of the same class) then
        return;
    for each attribute A do
        evaluate splits on attribute A;
    Use best split found to partition S into S1 and S2;
    Partition(S1);
    Partition(S2);
```

Initial call: Partition(TrainingData)

Figure 2: General Tree-growth Algorithm

There are two major issues that have critical performance implications in the tree-growth phase:

1. How to find split points that define node tests.
2. Having chosen a split point, how to partition the data.

The well-known CART [3] and C4.5 [20] classifiers, for example, grow trees depth-first and repeatedly sort the data at every node of the tree to arrive at the best splits for numeric attributes. SLIQ, on the other hand, replaces this repeated sorting with one-time sort by using separate lists for each attribute (see [15] for details). SLIQ uses a data structure called a *class list* which must remain memory resident at all times. The size of this structure is proportional to the number of input records, and this is what limits the number of input records that SLIQ can handle.

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

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

Figure 3: Example of attribute lists

SPRINT addresses the above two issues differently from previous algorithms; it has no restriction on the size of input and yet is a fast algorithm. It shares with SLIQ the advantage of a one-time sort, but uses different data structures. In particular, there is no structure like the class list that grows with the size of input and needs to be memory-resident. We further discuss differences between SLIQ and SPRINT in Section 2.4, after we have described SPRINT.

2.1 Data Structures

Attribute lists

SPRINT initially creates an *attribute list* for each attribute in the data (see Figure 3). Entries in these lists, which we will call *attribute records*, consist of an attribute value, a class label, and the index of the record (*rid*) from which these value were obtained. Initial lists for continuous attributes are sorted by attribute value once when first created. If the entire data does not fit in memory, attribute lists are maintained on disk.

The initial lists created from the training set are associated with the root of the classification tree. As the tree is grown and nodes are split to create new children, the attribute lists belonging to each node are partitioned and associated with the children. When a list is partitioned, the order of the records in the list is preserved; thus, partitioned lists never require resorting. Figure 4 shows this process pictorially.

Histograms

For continuous attributes, two histograms are associated with each decision-tree node that is under consideration for splitting. These histograms, denoted as C_{above} and C_{below}, are used to capture the class distribution of the attribute records at a given node. As we will see, C_{below} maintains this distribution for attribute records that have already been processed, whereas C_{above} maintains it for those that have not.

Categorical attributes also have a histogram associated with a node. However, only one histogram is needed and it contains the class distribution for each value of the given attribute. We call this histogram a *count matrix*.

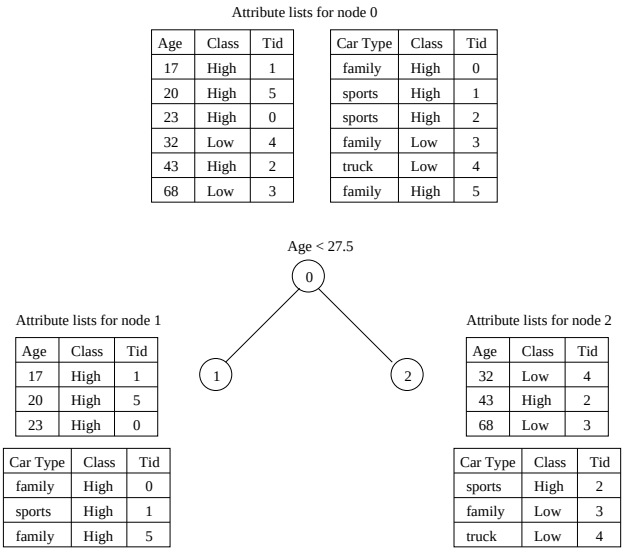


Figure 4: Splitting a node's attribute lists

Since attribute lists are processed one at a time, memory is required for only one set of such histograms. Figures 5 and 6 show example histograms.

2.2 Finding split points

While growing the tree, the goal at each node is to determine the split point that "best" divides the training records belonging to that leaf. The value of a split point depends upon how well it separates the classes. Several splitting indices have been proposed in the past to evaluate the goodness of the split. We use the *gini* index, originally proposed in [3], based on our experience with SLIQ. For a data set S containing examples from n classes, $gini(S)$ is defined as $gini(S) = 1 - \sum p_j^2$ where p_j is the relative frequency of class j in S. If a split divides S into two subsets S_1 and S_2, the index of the divided data $gini_{split}(S)$ is given by $gini_{split}(S) = \frac{n_1}{n} gini(S_1) + \frac{n_2}{n} gini(S_2)$. The advantage of this index is that its calculation requires only the distribution of the class values in each of the partitions.

To find the best split point for a node, we scan each of the node's attribute lists and evaluate splits based on that attribute. The attribute containing the split point with the lowest value for the gini index is then used to split the node. We discuss next how split points are evaluated within each attribute list.

Continuous attributes

For continuous attributes, the candidate split points are mid-points between every two consecutive attribute values in the training data. For determining the split for an attribute for a node, the histogram C_{below} is initialized to zeros whereas C_{above} is initialized with the class distribution for all the records for the node. For the root node, this distribution is obtained at the time of sorting. For other nodes this distribution is obtained when the node is created (discussed below in Section 2.3).

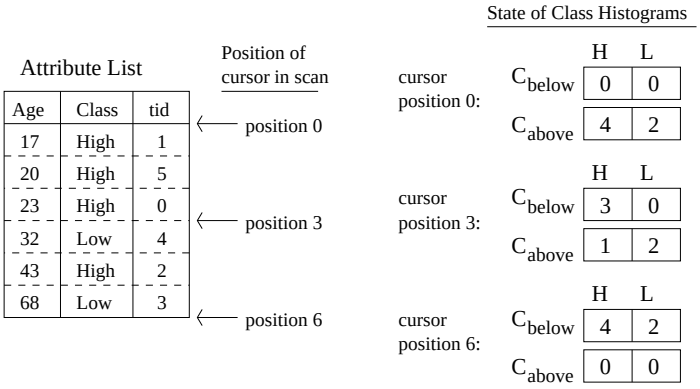


Figure 5: Evaluating continuous split points

Attribute records are read one at a time and C_{below} and C_{above} are updated for each record read. Figure 5 shows the schematic for this histogram update. After each record is read, a split between values (i.e. attribute records) we have and have not yet seen is evaluated. Note that C_{below} and C_{above} have all the necessary information to compute the gini index. Since the lists for continuous attributes are kept in sorted order, each of the candidate split-points for an attribute are evaluated in a single sequential scan of the corresponding attribute list. If a winning split point was found during the scan, it is saved and the C_{below} and C_{above} histograms are deallocated before processing the next attribute.

Categorical attributes

For categorical split-points, we make a single scan through the attribute list collecting counts in the count matrix for each combination of class label and attribute value found in the data. A sample of a count matrix after a data scan is shown in Figure 6. Once we are finished with the scan, we consider all subsets of the attribute values as possible split points and compute the corresponding gini index. If the cardinality of an attribute is above certain threshold, the greedy algorithm initially proposed for IND [17] is instead used for subsetting. The important point is that the information required for computing the gini index for any subset splitting is available in the count matrix.

The memory allocated for a count matrix is reclaimed after the splits for the corresponding attribute have been evaluated.

Attribute List

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

Count Matrix

	H	L
family	2	1
sports	2	0
truck	0	1

Figure 6: Evaluating categorical split points

2.3 Performing the split

Once the best split point has been found for a node, we execute the split by creating child nodes and dividing the attribute records between them. This requires splitting the node's lists for every attribute into two (see Figure 4 for an illustration)[2]. Partitioning the attribute list of the winning attribute (i.e. the attribute used in the winning split point — *Age* in our example) is straightforward. We scan the list, apply the split test, and move the records to two new attribute lists — one for each new child.

Unfortunately, for the remaining attribute lists of the node ($CarType$ in our example), we have no test that we can apply to the attribute values to decide how to divide the records. We therefore work with the *rids*. As we partition the list of the splitting attribute (i.e. *Age*), we insert the *rids* of each record into a probe structure (hash table), noting to which child the record was moved. Once we have collected all the *rids*, we scan the lists of the remaining attributes and probe the hash table with the *rid* of each record. The retrieved information tells us with which child to place the record.

If the hash-table is too large for memory, splitting is done in more than one step. The attribute list for the splitting attribute is partitioned upto the attribute record for which the hash table will fit in memory; portions of attribute lists of non-splitting attributes are partitioned; and the process is repeated for the remainder of the attribute list of the splitting attribute. If the hash-table can fit in memory (quite likely for nodes at lower levels of the tree), a simple optimization is possible. We can build the hash table out of the *rids* of only the smaller of the two children. Relative sizes of the two children are determined at the time the split point is evaluated.

During this splitting operation, we also build class histograms for each new leaf. As stated earlier, these histograms are used to initialize the C_{above} histograms when evaluating continuous split-points in the next pass.

[2]Because file-creation is usually an expensive operation, we have a solution that does not require the creation of new files for each new attribute list. The details of this optimization can be found in [22].

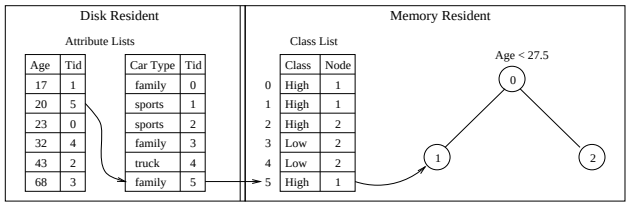


Figure 7: Attribute and Class lists in SLIQ

2.4 Comparison with SLIQ

The technique of creating separate attribute lists from the original data was first proposed by the SLIQ algorithm [15]. In SLIQ, an entry in an attribute list consists only of an attribute value and a *rid*; the class labels are kept in a separate data-structure called a *class list* which is indexed by *rid*. In addition to the class label, an entry in the class list also contains a pointer to a node of the classification tree which indicates to which node the corresponding data record currently belongs. Finally, there is only one list for each attribute. Figure 7 illustrates these data structures.

The advantage of not having separate sets of attribute lists for each node is that SLIQ does not have to rewrite these lists during a split. Reassignment of records to new nodes is done simply by changing the tree-pointer field of the corresponding class-list entry. Since the class list is randomly accessed and frequently updated, it *must* stay in memory all the time or suffer severe performance degradations. The size of this list also grows in direct proportion to the training-set size. This ultimately limits the size of the training set that SLIQ can handle.

Our goal in designing SPRINT was not to outperform SLIQ on datasets where a class list can fit in memory. Rather, the purpose of our algorithm is to develop an accurate classifier for datasets that are simply too large for any other algorithm, and to be able to develop such a classifier efficiently. Also, SPRINT is designed to be easily parallelizable as we will see in the next section.

3 Parallelizing Classification

We now turn to the problem of building classification trees in parallel. We again focus only on the growth phase due to its data-intensive nature. The pruning phase can easily be done off-line on a serial processor as it is computationally inexpensive, and requires access to only the decision-tree grown in the training phase.

In parallel tree-growth, the primary problems remain finding good split-points and partitioning the data using the discovered split points. As in any parallel algorithm, there are also issues of data placement and workload balancing that must be considered. Fortunately, these issues are easily resolved in the SPRINT algorithm. SPRINT was specifically designed to remove any dependence on data structures that are either centralized or memory-resident; because of these design goals, SPRINT parallelizes quite naturally and efficiently. In this section we will present how we parallelize SPRINT. For comparison, we also discuss two parallelizations of SLIQ.

These algorithms all assume a shared-nothing parallel environment where each of N processors has private memory and disks. The processors are connected by a communication network and can communicate only by passing messages. Examples of such parallel machines include GAMMA [7], Teradata [23], and IBM's SP2 [12].

3.1 Data Placement and Workload Balancing

Recall that the main data structures used in SPRINT are the attribute lists and the class histograms. SPRINT achieves uniform data placement and workload balancing by distributing the attribute lists evenly over N processors of a shared-nothing machine. This allows each processor to work on only $1/N$ of the total data.

The partitioning is achieved by first distributing the training-set examples equally among all the processors. Each processor then generates its own attribute-list partitions in parallel by projecting out each attribute from training-set examples it was assigned. Lists for categorical attributes are therefore evenly partitioned and require no further processing. However, continuous attribute lists must now be sorted and repartitioned into contiguous sorted sections. For this, we use the parallel sorting algorithm given in [8]. The result of this sorting operation is that each processor gets a fairly equal-sized sorted sections of each attribute list. Figure 3 shows an example of the initial distribution of the lists for a 2-processor configuration.

3.2 Finding split points

Finding split points in parallel SPRINT is very similar to the serial algorithm. In the serial version, processors scan the attribute lists either evaluating split-points for continuous attributes or collecting distribution counts for categorical attributes. This does not change in the parallel algorithm — no extra work or communication is required while each processor is scanning its attribute-list partitions. We get the full advantage of having N processors simultaneously and independently processing $1/N$ of the total data. The differences between the serial and parallel algorithms appear only before and after the attribute-list partitions are scanned.

Processor 0

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

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

Processor 1

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

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

Figure 8: Parallel Data Placement

Continuous attributes

For continuous attributes, the parallel version of SPRINT differs from the serial version in how it initializes the C_{below} and C_{above} class-histograms. In a parallel environment, each processor has a separate contiguous section of a "global" attribute list. Thus, a processor's C_{below} and C_{above} histograms must be initialized to reflect the fact that there are sections of the attribute list on other processors. Specifically, C_{below} must initially reflect the class distribution of all sections of an attribute-list assigned to processors of lower rank. The C_{above} histograms must likewise initially reflect the class distribution of the local section as well as all sections assigned to processors of higher rank. As in the serial version, these statistics are gathered when attribute lists for new leaves are created. After collecting statistics, the information is exchanged between all the processors and stored with each leaf, where it is later used to initialize that leaf's C_{above} and C_{below} class histograms.

Once all the attribute-list sections of a leaf have been processed, each processor will have what it considers to be the best split for that leaf. The processors then communicate to determine which of the N split points has the lowest cost.

Categorical attributes

For categorical attributes, the difference between the serial and parallel versions arises after an attribute-list section has been scanned to build the count matrix for a leaf. Since the count matrix built by each processor is based on "local" information only, we must exchange these matrices to get the "global" counts. This is done by choosing a coordinator to collect the count matrices from each processor. The coordinator process then sums the local matrices to get the global count-matrix.

As in the serial algorithm, the global matrix is used to find the best split for each categorical attribute.

3.3 Performing the Splits

Having determined the winning split points, splitting the attribute lists for each leaf is nearly identical to the serial algorithm with each processor responsible for splitting its own attribute-list partitions. The only additional step is that before building the probe structure, we will need to collect *rids* from all the processors. (Recall that a processor can have attribute records belonging to any leaf.) Thus, after partitioning the list of a leaf's splitting attribute, the *rids* collected during the scan are exchanged with all other processors. After the exchange, each processor continues independently, constructing a probe-structure with all the *rids* and using it to split the leaf's remaining attribute lists.

No further work is needed to parallelize the SPRINT algorithm. Because of its design, SPRINT does not require a complex parallelization and, as we will see in Section 4.3, scales quite nicely.

3.4 Parallelizing SLIQ

The attribute lists used in SLIQ can be partitioned evenly across multiple processors as is done in parallel SPRINT. However, the parallelization of SLIQ is complicated by its use of a centralized, memory-resident data-structure — the class list. Because the class list requires random access and frequent updating, parallel algorithms based on SLIQ require that the class list be kept memory-resident. This leads us to two primary approaches for parallelizing SLIQ : one where the class list is replicated in the memory of every processor, and the other where it is distributed such that each processor's memory holds only a portion of the entire list.

3.4.1 Replicated Class List

In the first approach, which we call SLIQ/R, the class list for the entire training set is replicated in the local memory of every processor. Split-points are evaluated in the same manner as in parallel SPRINT, by exchanging count matrices and properly initializing the class histograms. However, the partitioning of attribute lists according to a chosen split point is different.

Performing the splits requires updating the class list for each training example. Since every processor must maintain a consistent copy of the entire class list, every class-list update must be communicated to and applied by every processor. Thus, the time for this part of tree growth will increase with the size of the training set, even if the amount of data at each node remains fixed.

Although SLIQ/R parallelizes split-point evaluation and class-list updates, it suffers from the same drawback as SLIQ — the size of the training set is limited by the memory size of a single processor. Since each processor has a full copy of the class list, SLIQ/R can efficiently process a training set only if the class list for the entire database can fit in the memory of every processor. This is true regardless of the number of processors used.

3.4.2 Distributed Class List

Our second approach to parallelizing SLIQ, called SLIQ/D, helps to relieve SLIQ 's memory constraints by partitioning the class list over the multiprocessor. Each processor therefore contains only $1/Nth$ of the class list. Note that the partitioning of the class list has no correlation with the partitioning of the continuous attribute lists; the class label corresponding to an attribute value could reside on a different processor. This implies that communication is required to look up a "non-local" class label. Since the class list is created from the original partitioned training-set, it will be perfectly correlated with categorical attribute lists. Thus, communication is only required for continuous attributes.

Given this scenario, SLIQ/D has high communication costs while evaluating continuous split points. As each attribute list is scanned, we need to look-up the corresponding class label and tree-pointer for each attribute value. This implies that each processor will require communication for $N - 1/N$ of its data. Also, each processor will have to service lookup requests from other processors in the middle of scanning its attribute lists. Although our SLIQ/D implementation reduces the communication costs by batching the look-ups to the class lists, the extra computation that each processor performs in requesting and servicing remote look-ups to the class list is still high. SLIQ/D also incurs similar communication costs when the class list is updated while partitioning the data using the best splits found.

4 Performance Evaluation

The primary metric for evaluating classifier performance is *classification accuracy* — the percentage of *test* samples that are correctly classified. The other important metrics are *classification time* and the *size* of the decision tree. The ideal goal for a decision tree classifier is to produce compact, accurate trees in a short classification time.

Although the data structures and how a tree is grown are very different in SPRINT and SLIQ, they consider the same types of splits at every node and

use identical splitting index (gini index). The two algorithms, therefore, produce identical trees for a given dataset (provided SLIQ can handle the dataset). Since SPRINT uses SLIQ 's pruning method, the final trees obtained using the two algorithms are also identical. Thus, the accuracy and tree size characteristics of SPRINT are identical to SLIQ. A detailed comparison of SLIQ's accuracy, execution time, and tree size with those of CART [3] and C4 (a predecessor of C4.5 [20]) is available in [15]. This performance evaluation shows that compared to other classifiers, SLIQ achieves comparable or better classification accuracy, but produces small decision trees *and* has small execution times. We, therefore, focus only on the classification time metric in our performance evaluation in this paper.

4.1 Datasets

An often used benchmark in classification is STATLOG[16]; however, its largest dataset contains only 57,000 training examples. Due to the lack of a classification benchmark containing large datasets, we use the synthetic database proposed in [2] for all of our experiments. Each record in this synthetic database consists of nine attributes four of which are shown in Table 1. Ten classification functions were also proposed in [2] to produce databases with distributions with varying complexities. In this paper, we present results for two of these function. Function 2 results in fairly small decision trees, while function 7 produces very large trees. Both these functions divide the database into two classes: Group A and Group B. Figure 9 shows the predicates for Group A are shown for each function.

Function 2 - Group A:
$((\textbf{age} < 40) \wedge (50K \leq \textbf{salary} \leq 100K)) \vee$
$((40 \leq \textbf{age} < 60) \wedge (75K \leq \textbf{salary} \geq 125K)) \vee$
$((\textbf{age} \geq 60) \wedge (25K \leq \textbf{salary} \leq 75K))$

Function 7 - Group A:
$\textbf{disposable} > 0$
where $\textbf{disposable} = (0.67 \times (\textbf{salary} + \textbf{commission})) - (0.2 \times \textbf{loan} - 20K)$

Figure 9: Classification Functions for Synthetic Data

Table 1: Description of Attributes for Synthetic Data

Attribute	Value
salary	uniformly distributed from 20k to 150k
commission	**salary** $\geq 75k \Rightarrow$ **commission** $= 0$ else uniformly distributed from 10k to 75k
age	uniformly distributed from 20 to 80
loan	uniformly distributed from 0 to 500k

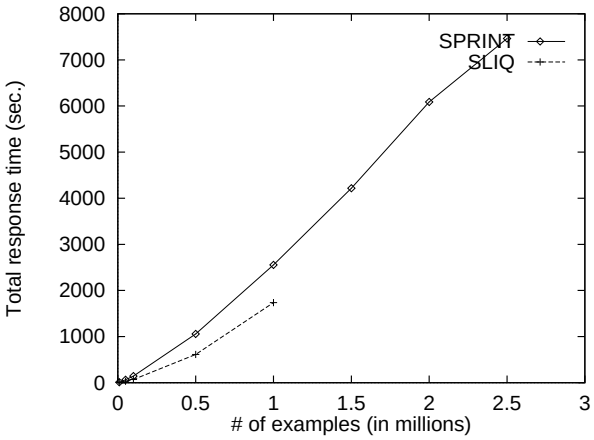


Figure 10: Response times for serial algorithms

4.2 Serial Performance

For our serial analysis, we compare the response times of serial SPRINT and SLIQ on training sets of various sizes. We only compare our algorithm with SLIQ because is has been shown in [15] that SLIQ in most cases outperforms other popular decision-tree classifiers. For the disk-resident datasets which we will be exploring here, SLIQ is the only other viable algorithm.

Experiments were conducted on an IBM RS/6000 250 workstation running AIX level 3.2.5. The CPU has a clock rate of 66MHz and 16MB of main memory. Apart from the standard UNIX daemons and system processes, experiments were run on an idle system.

We used training sets ranging in size from 10,000 records to 2.5 million records. This range was selected to examine how well SPRINT performs in operating regions where SLIQ can and cannot run. The results are shown in Figure 10 on databases generated using function 2.

The results are very encouraging. As expected, for data sizes for which the class list could fit in memory, SPRINT is somewhat slower than SLIQ. In this operating region, we are pitting SPRINT 's rewriting of the dataset to SLIQ 's in-memory updates to the class list. What is surprising is that even in this region SPRINT comes quite close to SLIQ. However, as soon as we cross an input size threshold (about 1.5 million records for our system configuration), SLIQ starts thrashing, whereas SPRINT continues to exhibit a nearly linear scaleup.

4.3 Parallel Performance

To examine how well the SPRINT algorithm performs in parallel environments, we implemented its parallelization on an IBM SP2 [12], using the standard MPI communication primitives [10]. The use of MPI allows our implementation to be completely portable to other

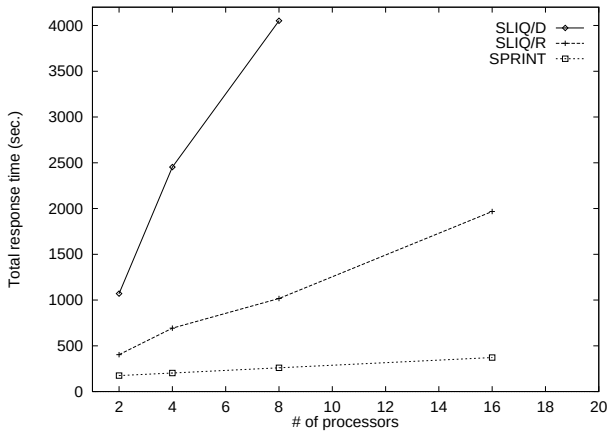


Figure 11: Response times for parallel algorithms

shared-nothing parallel architectures, including workstation clusters. Experiments were conducted on a 16-node IBM SP2 Model 9076. Each node in the multiprocessor is a 370 Node consisting of a POWER1 processor running at 62.5MHZ with 128MB of real memory. Attached to each node is a 100MB disk on which we stored our datasets. The processors all run AIX level 4.1 and communicate with each other through the High-Performance-Switch with HPS-tb2 adaptors. See [12] for SP2 hardware details.

Due to the available disk space being smaller than the available memory, we are prevented from running any experiments where attribute lists are forced to disk. This results in I/O costs, which scale linearly in SPRINT, becoming a smaller fraction of the overall execution time. Any other costs that may not scale well will thus be exaggerated.

4.3.1 Comparison of Parallel Algorithms

We first compare parallel SPRINT to the two parallelizations of SLIQ. In these experiments, each processor contained 50,000 training examples and the number of processors varied from 2 to 16. The total training-set size thus ranges from 100,000 records to 1.6 million records. The response times[3] for each algorithm are shown in Figure 11. To get a more detailed understanding of each algorithm's performance, we show in Figure 12 a breakdown of total response time into time spent discovering split points and time spent partitioning the data using the split points.

Immediately obvious is how poorly SLIQ/D performs relative to both SLIQ/R and SPRINT. The communication costs of using a distributed class-list and time spent servicing class-list requests from other processors are extremely high — so much so that SLIQ/D will probably never be an attractive algorithm despite its ability to handle training sets that are too large for either SLIQ and SLIQ/R. As shown in Figure 12, SLIQ/D pays this high penalty in both components of tree growth (i.e. split-point discovery and data partitioning) and scales quite poorly.

SPRINT performs much better than SLIQ/R, both in terms of response times and scalability. For both algorithms, finding the best split points takes roughly constant time, because the amount of data on each processor remains fixed as the problem size is increased. The increase in response times is from time spent partitioning the data. SPRINT shows a slight increase because of the cost of building the *rid* hash-tables used to split the attribute lists. Since these hash-tables may potentially contain the *rids* of all the tuples belonging to a particular leaf-node, this cost increases with the data size. SLIQ/R performs worse than SPRINT, because each processor in SLIQ/R must not only communicate but also apply class-list updates for every training example. As the problem size increase, so do the number of updates each processor must perform. While SPRINT may perform as much communication as SLIQ/R, it only requires processors to update their own local records.

The rest of this section examines the scalability, speedup, and sizeup characteristics of SPRINT in greater detail.

4.3.2 Scaleup

For our first set of sensitivity experiments, each processor has a fixed number of training examples and we examined SPRINT's performance as the configuration changed from 2 to 16 processors. We studied three of these scaleup experiments, with 10, 50 and 100 thousand examples on each processor. The results of these runs are shown in Figure 13. Since the amount of data per processor does not change for a given experiment, the response times should ideally remain constant as the configuration size is increased.

The results show nice scaleup. The drop in scaleup is due to the time needed to build SPRINT's *rid* hash-tables. While the amount of local data on each processor remains constant, the size of these hash-tables does not. The *rid* hash-tables grow in direct proportion to the total training-set size. Overall, we can conclude that parallel SPRINT can indeed be used to classify very large datasets.

4.3.3 Speedup

Next, we examined the speedup characteristics of SPRINT. We kept the total training set constant and changed the processor configuration. We did this for training-set sizes of 800 thousand and 1.6 million examples. Results for these speedup experiments are

[3] Response time is the total real time measured from the start of the program until its termination.

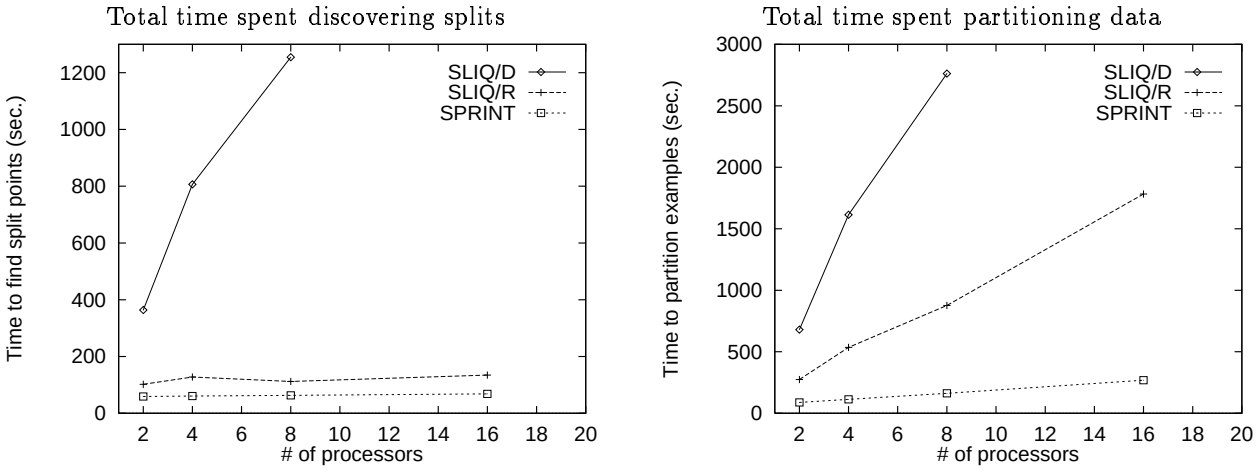


Figure 12: Breakdown of response times

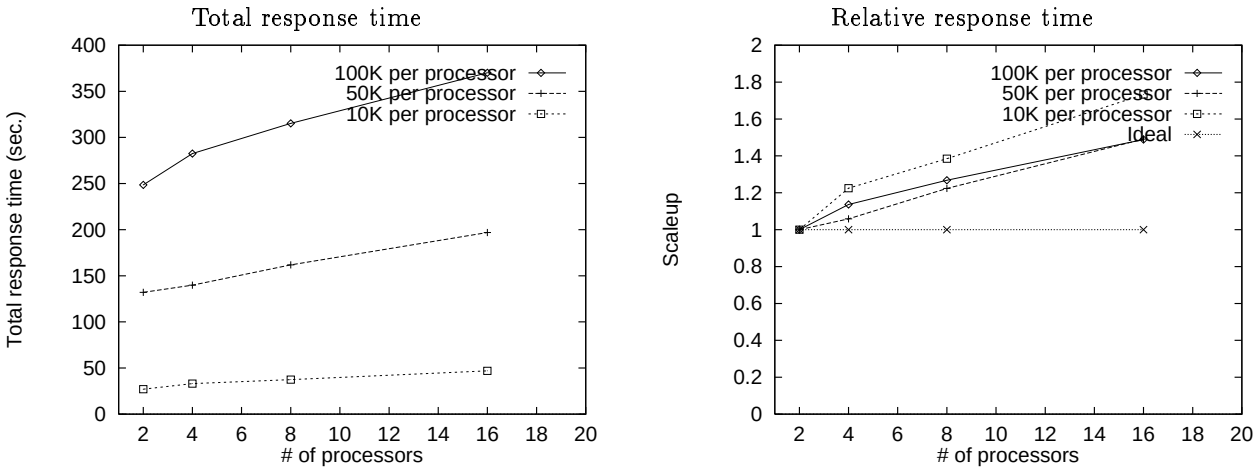


Figure 13: Scaleup of SPRINT

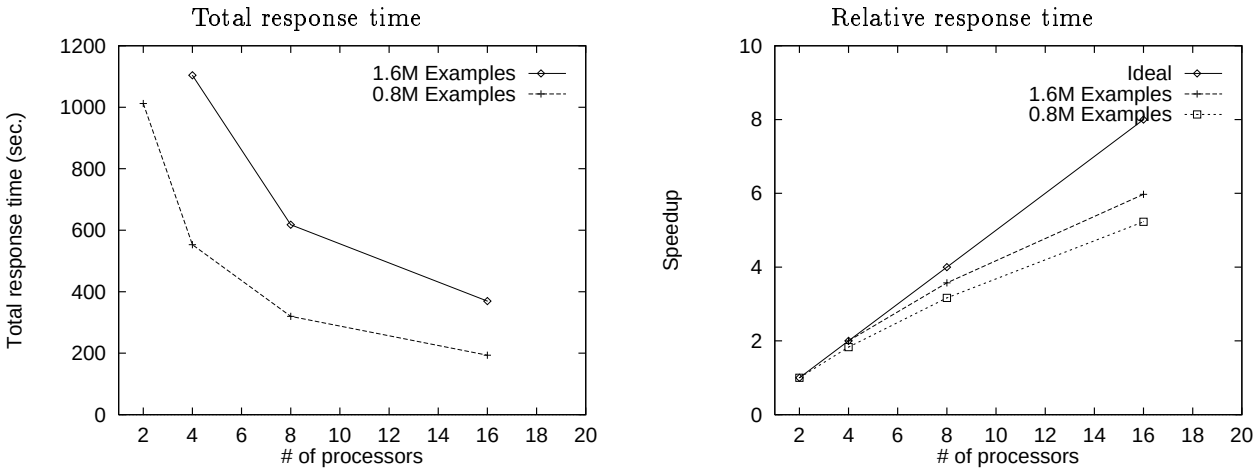


Figure 14: Speedup of SPRINT

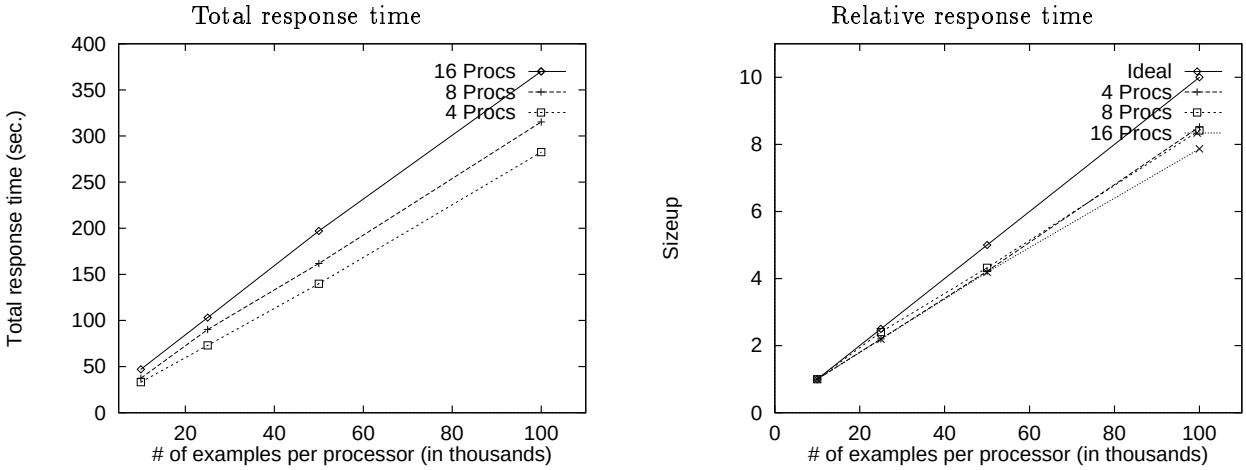


Figure 15: Sizeup of SPRINT

shown in Figure 14. Due to limited disk space, the 2-processor configuration could not create the dataset containing the 1.6 million examples. As can be expected, speedup performance improves with larger datasets. For small datasets, communication becomes a significant factor of the overall response time. This is especially true as the configuration sizes are increased to the point where there are only a few tens of thousand examples on each processor. Another factor limiting speedup performance is the *rid* hash-tables. These hash tables have the same size regardless of the processor configuration. Building these hash-tables thus requires a constant amount of time whether we are using 2 or 16 processors. These experiments show that we do get nice speed-up with SPRINT, with the results improving for larger datasets.

4.3.4 Sizeup

In sizeup experiments, we examine how SPRINT performs on a fixed processor configuration as we increase the size of the dataset. Figure 15 shows this for three different processor configurations where the per-processor training-set size is increased from 10 thousand to 100 thousand examples. SPRINT exhibits sizeup results better than ideal — processing twice as much data does not require twice as much processing time. The reason is that communication costs for exchanging split points and count matrices does not change as the training-set size is increased. Thus, while doubling the training-set size doubles most of the response costs, others remain unaffected. The result is superior sizeup performance.

5 Conclusion

With the recent emergence of the field of data mining, there is a great need for algorithms for building classifiers that can handle very large databases. The recently proposed SLIQ algorithm was the first to address these concerns. Unfortunately, due to the use of a memory-resident data structure that scales with the size of the training set, even SLIQ has an upper limit on the number of records it can process.

In this paper, we presented a new classification algorithm called SPRINT that removes all memory restrictions that limit existing decision-tree algorithms, and yet exhibits the same excellent scaling behavior as SLIQ. By eschewing the need for any centralized, memory-resident data structures, SPRINT efficiently allows classification of virtually any sized dataset. Our design goals also included the requirement that the algorithm be easily and efficiently parallelizable. SPRINT does have an efficient parallelization that requires very few additions to the serial algorithm.

Using measurements from actual implementations of these algorithms, we showed that SPRINT is an attractive algorithm in both serial and parallel environments. On a uniprocessor, SPRINT exhibits execution times that compete favorably with SLIQ. We also showed that SPRINT handles datasets that are too large for SLIQ to handle. Moreover, SPRINT scales nicely with the size of the dataset, even into the large problem regions where no other decision-tree classifier can compete.

Our implementation on SP2, a shared-nothing multiprocessor, showed that SPRINT does indeed parallelize efficiently. It outperforms our two parallel implementations of SLIQ in terms of execution time and scalability. Parallel SPRINT's efficiency improves as the problem size increases. It has excellent scaleup, speedup, and sizeup characteristics.

Given SLIQ's somewhat superior performance in problem regions where a class list can fit in memory, one can envision a hybrid algorithm combining SPRINT and SLIQ. The algorithm would initially run SPRINT until a point is reached where a class list

could be constructed and kept in real memory. At this point, the algorithm would switch over from SPRINT to SLIQ exploiting the advantages of each algorithm in the operating regions for which they were intended. Since the amount of memory needed to build a class list is easily calculated, the switch over point would not be difficult to determine. We plan to build such a hybrid algorithm in future.

References

[1] Rakesh Agrawal, Sakti Ghosh, Tomasz Imielinski, Bala Iyer, and Arun Swami. An interval classifier for database mining applications. In *Proc. of the VLDB Conference*, pages 560–573, Vancouver, British Columbia, Canada, August 1992.

[2] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914–925, December 1993.

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

[4] Jason Catlett. *Megainduction: Machine Learning on Very Large Databases.* PhD thesis, University of Sydney, 1991.

[5] Philip K. Chan and Salvatore J. Stolfo. Experiments on multistrategy learning by meta-learning. In *Proc. Second Intl. Conference on Info. and Knowledge Mgmt.*, pages 314–323, 1993.

[6] Philip K. Chan and Salvatore J. Stolfo. Meta-learning for multistrategy and parallel learning. In *Proc. Second Intl. Workshop on Multistrategy Learning*, pages 150–165, 1993.

[7] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The Gamma database machine project. In *IEEE Transactions on Knowledge and Data Engineering*, pages 44–62, March 1990.

[8] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Proc. of the 1st Int'l Conf. on Parallel and Distributed Information Systems*, pages 280–291, December 1991.

[9] D. J. Fifield. Distributed tree construction from large data-sets. Bachelor's Honours Thesis, Australian National University, 1992.

[10] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, May 1994.

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

[12] Int'l Business Machines. *Scalable POWERparallel Systems*, GA23-2475-02 edition, February 1995.

[13] M. James. *Classificaton Algorithms.* Wiley, 1985.

[14] R. Lippmann. An introduction to computing with neural nets. *IEEE ASSP Magazine*, 4(22), April 1987.

[15] Manish Mehta, Rakesh Agrawal, and Jorma Rissanen. SLIQ: A fast scalable classifier for data mining. In *Proc. of the Fifth Int'l Conference on Extending Database Technology (EDBT)*, Avignon, France, March 1996.

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

[17] NASA Ames Research Center. *Introduction to IND Version 2.1*, GA23-2475-02 edition, 1992.

[18] J. R. Quinlan. Induction over large databases. Technical Report STAN-CS-739, Stanford University, 1979.

[19] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[20] J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufman, 1993.

[21] J. Rissanen. *Stochastic Complexity in Statistical Inquiry.* World Scientific Publ. Co., 1989.

[22] John C. Shafer, Rakesh Agrawal, and Manish Mehta. SPRINT: A scalable parallel classifier for data
mining. Research report, IBM Almaden Research Center, San Jose, California, 1996. Available from `http://www.almaden.ibm.com/cs/quest`.

[23] Teradata Corp. *DBC/1012 Data Base Computer System Manual*, C10-0001-02 release 2.0 edition, November 1985.

[24] Sholom M. Weiss and Casimir A. Kulikowski. *Computer Systems that Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems.* Morgan Kaufman, 1991.

[25] J. Wirth and J. Catlett. Experiments on the costs and benefits of windowing in ID3. In *5th Int'l Conference on Machine Learning*, 1988.