ScalParC : A New Scalable and Efficient Parallel Classification Algorithm for Mining Large Datasets *

Mahesh V. Joshi George Karypis Vipin Kumar

Department of Computer Science
University of Minnesota, Minneapolis, MN 55455.
{mjoshi,karypis,kumar}@cs.umn.edu

Abstract

In this paper, we present ScalParC (Scalable Parallel Classifier), a new parallel formulation of a decision tree based classification process. Like other state-of-the-art decision tree classifiers such as SPRINT, ScalParC is suited for handling large datasets. We show that existing parallel formulation of SPRINT is unscalable, whereas ScalParC is shown to be scalable in both runtime and memory requirements. We present the experimental results of classifying up to 6.4 million records on up to 128 processors of Cray T3D, in order to demonstrate the scalable behavior of ScalParC. A key component of ScalParC is the parallel hash table. The proposed parallel hashing paradigm can be used to parallelize other algorithms that require many concurrent updates to a large hash table.

1 Introduction

Classification is an important problem in the rapidly emerging field of data mining. The problem can be stated as follows. We are given a *training dataset* consisting of records. Each record is identified by a unique *record id* and consists of fields corresponding to the *attributes*. An attribute with a continuous domain is called a *continuous* attribute. An attribute with finite domain of discrete values is called a *categorical* attribute. One of the categorical attributes is the *classifying attribute or class* and the values in its domain are called *class labels*. Classification is the process of discovering a model for the class in terms of the remaining attributes. The decision-tree models are found to be most useful in the domain of data mining. They yield comparable or better accuracy as compared to other models such as neural networks, statistical models or genetic models [8]. The decision-tree based classifiers that handle large datasets are attractive, because use of larger datasets improves the classification accuracy even further[2]. Recently proposed classifiers SLIQ [7] and SPRINT [10] use entire dataset for classification and are shown to be more accurate as compared to the classifiers that use sampled dataset or multiple partitions of the dataset [2, 3].

The decision tree model is built by recursively splitting the training set based on a locally optimal criterion until all or most of the records belonging to each of the partitions bear the same class label. Briefly, there are two phases to this process at each node of the decision tree. First phase determines the splitting decision and second phase splits the data. The very difference in the nature of continuous and categorical attributes requires them to be handled in different manners. The handling of categorical attributes in both phases is straightforward. Handling the continuous attributes is challenging. An efficient determination of the splitting decision used in most of the existing classifiers requires these attributes to be sorted on values. The classifiers such as CART [1] and C4.5 [9] perform sorting at every node of the decision tree, which makes them very expensive for large datasets, since this sorting has to be done out-of-core. The approach taken by SLIQ and SPRINT sorts the continuous attributes only once in the beginning. The splitting phase maintains this sorted order without requiring to sort the records again. The attribute lists are split in a consistent manner using a mapping between a record identifier and the node to which it belongs after splitting. SPRINT implements this mapping as a hash table, which is built on-the-fly for every node of the decision tree. The size of this hash table is proportional to the number of records at the node. For

*This work was supported by NSF CCR-9423082, by Army Research Office contract DA/DAAH04-95-1-0538, by Army High Performance Computing Research Center cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008, the content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred. Access to computing facilities was provided by AHPCRC, Minnesota Supercomputer Institute. Related papers are available via WWW at URL: http://www.cs.umn.edu/~kumar.

the upper levels of the tree, this number is $O(N)$, where N is the number of records in the training set. If the hash table does not fit in the main memory, then SPRINT has to divide the splitting phase into several stages such that the hash table for each of the phases fits in the memory. This requires multiple passes over each of the attribute lists causing expensive disk I/O.

The memory limitations faced by serial decision-tree classifiers and the need of classifying much larger datasets in shorter times make the classification algorithm an ideal candidate for parallelization. The parallel formulation, however, must address the issues of efficiency and scalability in both memory requirements and parallel runtime. Relatively little work has been done so far in developing parallel formulations of decision tree based classifiers [10, 4]. Among these, the most relevant one is the parallel formulation of SPRINT[10], as it requires sorting of continuous attributes only once. SPRINT's design allows it to parallelize the split determining phase effectively. The parallel formulation proposed for the splitting phase, however, is inherently unscalable in both memory requirements and runtime. It builds the required hash table on *all* the processors by gathering the record-id-to-node mapping from all the processors. For this phase, the communication overhead per processor is $O(N)$. Apart from the initial sorting phase, the serial runtime of a classifier is $O(N)$. Hence, SPRINT is unscalable in runtime. It is unscalable in memory requirements also, because the memory requirement per processor is $O(N)$, as the size of the hash table is of the same order as the size of the training dataset for the upper levels of the decision tree, and it resides on every processor.

In this paper, we present a new parallel formulation of a decision tree based classifier. We call it ScalParC (*Scal*able *Par*allel *C*lassifier) because it is *truly* scalable in both runtime and memory requirements. Like SPRINT, ScalParC sorts the continuous attributes only once in the beginning. It uses attribute lists similar to SPRINT. The key difference is that it employs a distributed hash table to implement the splitting phase. The communication structure used to construct and access this hash table introduces a new parallel hashing paradigm. A detailed analysis of applying this paradigm to the splitting phase shows that the overall communication overhead of the phase does not exceed $O(N)$, and the memory required to implement the phase does not exceed $O(N/p)$ per processor. This makes ScalParC scalable in both runtime and memory requirements. We implemented the algorithm using MPI to make it portable across most of today's parallel machines. We present the experimental results of classifying up to 6.4 million records on up to 128 processors of Cray T3D, in order to demonstrate the scalable behavior of ScalParC in both runtime and memory requirements.

The paper is organized as follows. We describe the sequential decision tree based classification process in detail, and identify the limitations of serial algorithms in section 2. Section 3 gives a detailed description of the ScalParC design process, by discussing the issues of load balancing, data distribution, possible parallelization approaches, the parallel hashing paradigm and its application to the splitting phase. Section 4 outline the algorithm. The experimental results are presented in Section 5.

2 Sequential Decision Tree based Classification

A decision tree model consists of internal nodes and leaves. Each of the internal nodes has a decision associated with it and each of the leaves has a class label attached to it. A decision-tree based classification learning consists of two steps. In the first step of *tree induction*, a tree is induced from the given training set. In the second step of *tree pruning*, the induced tree is made more concise and robust by removing any statistical dependencies on the specific training dataset. The induction step is computationally much more expensive as compared to the pruning step. In this paper, we concentrate only on the induction step.

Tree induction consists of two phases at each of the internal nodes. First phase makes a splitting decision based on optimizing a splitting index. We call this phase the *split determining phase*. The second phase is called the *splitting phase*. It splits the records into children nodes based on the decision made. The process stops when all the leaves have records bearing only one class label.

One of the commonly used splitting criteria is to minimize the gini index [1] of the split. The calculation of gini index involves computing the frequency of each class in each of the partitions. Let a parent node, having n records from c possible classes, be split into d partitions, each partition corresponding to a child of the the parent node. The gini index for 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, among which n_{ij} records bear class label j. The matrix $[n_{ij}]$ is called the *count matrix*. The gini index of the total split is given by $gini_{split} = \sum_{i=1}^{d}(n_i/n)\ gini_i$. The partitions are formed based on a *splitting decision* which consists of a condition on the values of a single attribute called the *splitting attribute*. The condition which gives the least value for $gini_{split}$ is chosen to split the records at that node.

Each of the attributes is a candidate for being the splitting attribute. For a continuous attribute, A, we assume that two partitions are formed based on the condition $A < v$, for some value v in its domain. One partition contains records which satisfy the condition and the other contains the rest. For a categorical attribute, B, having m distinct values in its domain, we assume that the splitting decision forms m

partitions[1], one for each of the values of B. The computation of gini index for a categorical attribute is straightforward because there is only one count matrix possible. For a continuous attribute, we need to decide on the value v. If the continuous attribute is sorted on its values at each of the nodes in the tree, then a linear search can be made for the optimal value of v by moving the possible split point from the beginning to the end of the list, one value at a time, updating the count matrix, and computing the $gini_{split}$ value for each point. Since each attribute needs to be processed separately, the training set is fragmented vertically into lists for each of the attributes. These lists are called the *attribute lists*. A record id is associated with each value in all the attribute lists.

After the splitting decision is made, the second phase splits the records among the children nodes. All the attribute lists should be split. The information regarding which record gets assigned to which node is obtained based on the splitting decision and the record ids in the list of the splitting attribute. With this information, the list of splitting attribute can be split easily. The lists of other attributes must be split consistently, which means that the values belonging to a particular record id from all the attribute lists must get assigned to the same node of the decision tree. Different lists, in general, may have a different order on record ids, because the continuous attributes are sorted on values. Hence to perform a consistent assignment efficiently, some kind of a mapping of record ids to node is required. The structure of this mapping is determined by the approach used for splitting. Among the different possible approaches, SPRINT's approach is the most suitable one for handling large datasets on a serial machine.

SPRINT associates the class information along with the record id for each value in the attribute lists. It splits each of the attribute lists physically among nodes. The splitting is done such that the continuous attribute lists maintain their sorted order on values at each of the nodes. All these choices make it possible to implement the split determining phase efficiently by a sequential scan of continuous attribute lists. Once the splitting decision is made, it is straightforward to split the list of the splitting attribute. In order to split other lists in a consistent manner, a hash table is formed to maintain a mapping of record ids to nodes. The hash table is built from record ids in the splitting attribute's list in conjunction with the splitting decision. Then the record ids in the lists of non-spitting attributes are searched to get the node information, and perform the split accordingly. The size of this hash table is proportional to the number of records at the current node. For the root node of the decision tree, this size is the same as the original training dataset size, and it remains of the same order for the nodes at upper levels. Thus, this approach also faces memory limitations for the upper levels of the tree. If the hash table does not fit in the memory, then multiple passes need to be done over the entire data requiring additional expensive disk I/O.

[1] It is also possible to form two partitions for a categorical attribute each characterized by a subset of values in its domain.

3 Designing ScalParC

The design goals for a parallel formulation of the decision tree based classification algorithms are scalability in both runtime and memory requirements. The parallel formulation should overcome the memory limitations faced by the sequential algorithms; i.e., it should make it possible to handle larger datasets without requiring redundant disk I/O. Also, a parallel formulation should offer good speedups over serial algorithms. In this section, we describe the design issues involved in parallelizing the classification process described in the previous section and propose the design approach taken by ScalParC.

The parallel runtime consists of computation time and the parallelization overhead. If T_s is the serial runtime of an algorithm and T_p is the parallel runtime on p processors, then the parallelization overhead is given by $T_o = pT_p - T_s$. For runtime scalability, the overhead, T_o, should not exceed $O(T_s)$[6]; i.e., the parallelization overhead per processor should not exceed $O(T_s/p)$.

For the classification problem at hand, let the training set size be N and the problem be solved on p processors. Let there be n_c classes and n_a attributes out of which n_t are continuous and n_g are categorical. After the initial sorting of the continuous attributes, the serial runtime is $T_s = O(N)$ for a majority of levels, when large datasets are being classified. Memory efficient and scalable formulations of parallel sorting are well known [6]. Hence, for runtime scalability, the algorithm must be designed such that none of the components of the overall communication overhead of the classification process exceeds $O(N)$ at any level; i.e., the per processor communication overhead should not exceed $O(N/p)$ per level. Since the memory required to solve the classification problem on a serial machine is $O(N)$, for memory scalability of the parallel formulation, the amount of memory per processor should not exceed $O(N/p)$.

3.1 Data Distribution and Load Balancing

The first design issue is the assignment of data to the processors. The computation time at a node is proportional to the number of records at that node. So, the best way to achieve load balance at the root node is to fragment each attribute list horizontally into p fragments of equal sizes and to assign each fragment to a different processor [10].

We assume that the initial assignment of data to the processors remains unchanged throughout the process of classification. With this approach, the splitting decision may

cause the records of a node to be distributed unevenly among processors. If the computations for *all* the records of all the nodes at a level are performed before doing any synchronizing communication between the processors, then the per-node imbalance would not be a concern. This approach of doing per-level communications as against per-node communications should also perform better on the parallel machines with high communication latencies, especially because the number of nodes will be large at the levels much deeper in tree.

3.2 Possible Parallelization Approaches

The next issue is the design of algorithms for each of the two phases of tree induction. The implementation of the split determining phase is straightforward. For the continuous attributes, the design presented in parallel formulation of SPRINT [10] is efficient. For a given continuous attribute, it initializes the count matrices on every processor corresponding to the split point lying at the beginning of the local attribute list on that processor. Then each processor proceeds to compute gini indices for all possible positions in its local part of the list. For the categorical attributes, the local count matrices from all the processors can be gathered onto a single processor, where the gini index can be computed.

The parallelization of the splitting phase is more difficult. In the parallel formulation of SPRINT, the required hash table is built on *all* the processors for each node of the decision tree. Then each processor splits its local copies of all the attribute lists as in the sequential algorithm. Since each processor has to receive the entire hash table, the amount of communication overhead per processor is proportional to the size of the hash table, which is $O(N)$ as noted earlier in section 2. Hence, this approach is not scalable in runtime. The approach is not scalable in terms of memory requirements also, because the hash table size on each processor is $O(N)$ for the top node as well as for nodes at the upper levels of the tree.

3.3 The ScalParC Parallelization Approach

Here, we present our approach to parallelizing the splitting phase which is scalable in both memory and runtime requirements.

3.3.1 Parallel Hashing Paradigm

We first present the scalable parallel hashing paradigm that is used in ScalParC to achieve scalability in the splitting phase. The paradigm gives mechanisms to construct and search a distributed hash table, when many values need to be hashed at the same time. We assume that there is a hash function, h, that hashes a given key, k, to yield a pair of numbers $h(k) = (p_i, l)$, such that k hashes to location l on the local part of the hash table residing on processor p_i. Each key k is associated with a value v. The hash table is constructed as follows. First each processor scans through all its (k, v) pairs and hashes k in each pair to determine the destination processor, p_i, and location, l, for storing value v. Every processor maintains a separate buffer destined for each of the processors, p_i. Each entry in this buffer is an (l, v) pair. Note that some of these buffers might be empty, if none of the keys hashes to the corresponding processors. Then one step of an all-to-all-personalized communication [6] is done. Finally, each processor extracts the (l, v) pairs from the received buffers and stores value v at index l of their respective hash tables. The same process can be followed while searching for values given their corresponding keys. Each processor hashes its keys, k, to fill an enquiry buffer for all the processors p_i with indices l. The receiving processors look up for v values at indices l, fill them in a buffer and another step of all-to-all-personalized communication gets the values back to the processors who require them. If each processor has m keys to hash at a time, then the all-to-all personalized communication can be done in $O(m)$ time provided m is $\Omega(p)$ [6]. Thus, the parallel hashing done in the proposed manner above is scalable as long as $\Omega(p^2)$ keys are hashed at the same time. Note that this paradigm can also support collisions by implementing open chaining at the indices l of the local hash tables.

3.3.2 Applying the Paradigm to ScalParC

Now, let us consider applying this paradigm to the splitting phase of classification. Once the splitting decision has been made at a node, a record-id-to-node mapping needs to be constructed using the list of splitting attribute, and then, this mapping will be inquired while splitting other attribute lists. The information needed to be stored in the mapping is the child number that a record belongs to after splitting. We call this mapping a *node table*. The node table is assumed to be a hash table with the hash function $h(j) = (j \; div \; N/p, j \; mod \; N/p)$, where the global record id j is the key and the child number is the value associated with it. The node table is of size N. It is distributed equally among p processors; hence, the size on each processor is $O(N/p)$. Note that, since the total number of global records is N, the above hash function is collision-free. Figure 1(b) shows a distributed node table for the example training dataset shown in the part (a) of the figure. For the example in the figure, we have $N = 9$ and $p = 3$, hence the hash function is $h(j) = (p_i, l)$, where $p_i = j \; div \; 3$ and $l = j \; mod \; 3$. Initially, this node table is empty. We now apply the communication structure described in section 3.3.1 to update and inquire the child number information stored in the distributed node table.

The process of updating the node table is similar to the

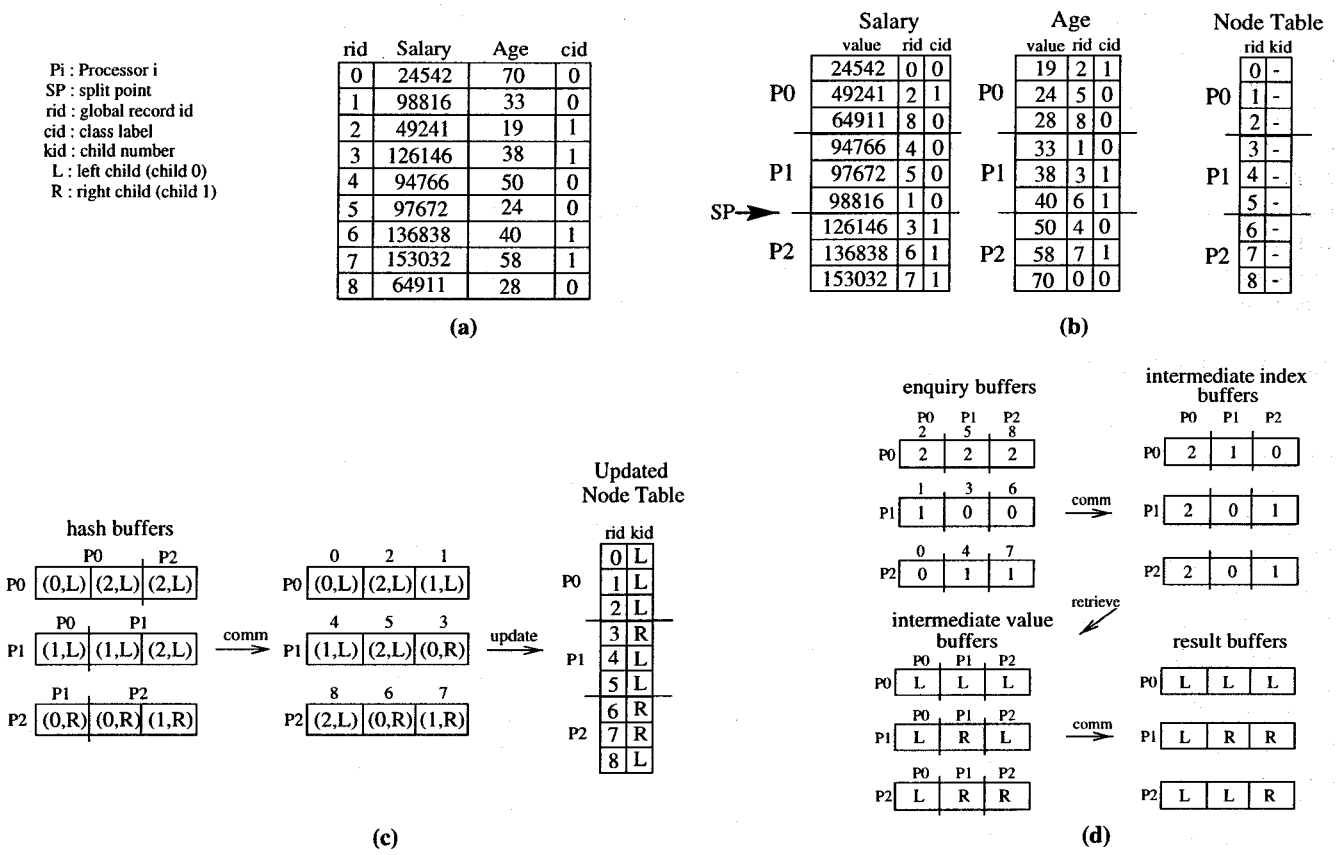

Figure 1. Illustrating the concept of distributed node table and the application of parallel hashing paradigm to splitting phase of ScalParC. (a) An example training set. (b) The attribute lists and empty node table at the beginning of splitting phase at the root node. (c) The process of hashing values into the node table. The global record id is shown corresponding to each of the entries in the received buffers. (d) The process of enquiring values from the node table. The global record id is shown corresponding to each of the entries in the enquiry buffers.

process of constructing a distributed hash table. Figure 1(c) illustrates this process at the top node of the decision tree being built using the training set of Figure 1(a). The splitting attribute is $Salary$ and the optimal split point is as shown. The record-id-to-node mapping obtained out of the list of splitting attribute is used to form the *hash buffers*. Each element of these buffers is a $(l, child\,number)$ pair. For example, after the splitting decision, processor $P0$ knows that all the record ids in its local list of $Salary$ belong to child 0 or the left child (denoted by L). So, after hashing record id 8, it fills an entry $(l = 8\,mod\,3 = 2, child\,number = L)$ in processor $P2$'s hash buffer. Each processor follows the same procedure. Then, a step of all-to-all-personalized communication is performed, and the node table is updated, using the received information. The updated node table is shown in the figure.

After updating the node table, it needs to be inquired for the child number information in order to split the lists of non-splitting attributes. Each attribute list is split separately. The process is illustrated in Figure 1(d) for the attribute Age at the root node. Using the paradigm above, each processor first forms the *enquiry buffers* by hashing the record ids in its local list of Age. An enquiry buffer for a processor contains local indices l. For example, processor $P1$ forms an enquiry buffer for processor $P2$ containing $l = 0$ after hashing the global record id 6 in its local list. After a step of all-to-all-personalized communication, each processor receives the *intermediate index buffers* containing local indices to be searched for. The node table is then searched, and the child numbers obtained are used to fill *intermediate value buffers*. These are communicated using another step of all-to-all-personalized communication to form the *result buffers*, which are used to extract the child information. All these buffers are shown in Figure 1(d) for the enquiry pro-

Pre-sort
$l = 1$ (corresponds to the root level of the decision tree)
do while (there are non-empty nodes at level l)
    Find-Split I
    Find-Split II
    Perform-Split-I
    Perform-Split-II
    $l = l+1$
end do

Figure 2. ScalParC tree induction algorithm.

cess of *Age*.

A detailed per-level analysis of runtime and memory scalability of the above process shows that no processor receives more than $O(N/p)$ updates to the node table, and no processor inquires more than $(n_a N/p)$ entries in the node table. There is a possibility, however, that some processors might send more than $O(N/p)$ updates to the node table. This can render the formulation runtime unscalable, but it can be shown that *no* parallelization approach, which sorts the continuous attributes and maintains initial data distribution, can achieve runtime scalability in such cases. The memory scalability is still ensured in ScalParC in such cases, by dividing the updates being sent into blocks of N/p. The detailed analysis and some possible ways of optimizing the communication overheads are given in [5].

To summarize, applying the parallel hashing paradigm makes ScalParC truly memory scalable. Furthermore, it is also runtime scalable except for some pathological cases.

4 The ScalParC Algorithm

The main data structures used in ScalParC are the distributed attribute lists, the distributed node table, and the count matrices. The details can be found in [5].

With these data structures, the ScalParC tree induction algorithm is shown in Figure 2.

In the Pre-sort phase, we use the scalable parallel sample sort algorithm [6] followed by a parallel shift operation, to sort all the continuous attributes.

In the Find-Split-I phase, for each continuous attribute, the local count matrix is computed for each node, corresponding to the split point lying at the beginning of the local attribute list, and then a parallel prefix operation is applied to compute the global count matrix for that split point position. For a categorical attribute, a processor is designated to coordinate the computation of the global count matrices for all the nodes, using parallel reduction operation [6].

In the Find-Split-II phase, termination criterion is applied to decide if a node needs further splitting. For the nodes requiring a split, the optimal gini index is computed using the global count matrices found in previous phase. For a continuous attribute, the local list is scanned one record at a time to find the optimal split point. For a categorical attribute, the designated processor computes the gini index. The overall best splitting criteria for each node is found using a parallel reduction operation.

In the Perform-Split-I phase, the lists of splitting attributes are split, the hash buffers are formed, and the distributed node table is updated using the process described in section 3.3.2. As noted in that section, there might be more than one communication steps needed to update the node table, in order to ensure memory scalability.

In the Perform-Split-II phase, the lists of all non-splitting attributes are split, one attribute at a time. For each such attribute, the node table is inquired using the enquiry process described in section 3.3.2. The collected node information is then used to split the attribute.

Refer to [5] for illustrations and further details of the algorithm.

5 Experimental Results

We have implemented the ScalParC tree induction algorithm using MPI. We tested it on up to 128 processors of a Cray T3D where each processor had 64MB of memory. We benchmarked the combination of Cray's tuned MPI implementation and the underlying communication subsystem assuming a linear model of communication. On an average, we obtained a latency of 100 μsec and bandwidth of 50 MB/sec for point-to-point communications, and a latency of 25 μsec per processor and bandwidth of 40 MB/sec for the all-to-all collective communication operations. We tested ScalParC for training sets containing up to 6.4 million records, each containing seven attributes. There were two possible class labels. The training sets were artificially generated using a scheme similar to that used in SPRINT[10].

Figure 3(a) shows the runtime scalability of ScalParC by plotting the speedup obtained for various training set sizes. For a given problem instance, the relative speedups decrease as the number of processors are increased, because of increased overheads. In particular, for 1.6 million records, ScalParC achieved a relative speedup of 1.61 while going from 16 to 32 processors, and a relative speedup of 1.31 while going from 64 to 128 processors. Relative speedups improve for larger problem sizes, because of increased computation to communication ratio. In particular, while going from 64 to 128 processors, the relative speedup obtained for 6.4 million records was 1.43 and a relative speedup obtained for 3.2 million records was 1.36. These trends are typical of a normal scalable parallel algorithm[6]. Note that ScalParC could classify 6.4 million records in just 77 seconds on 128 processors. This demonstrates that large classification problems can be solved quickly using ScalParC.

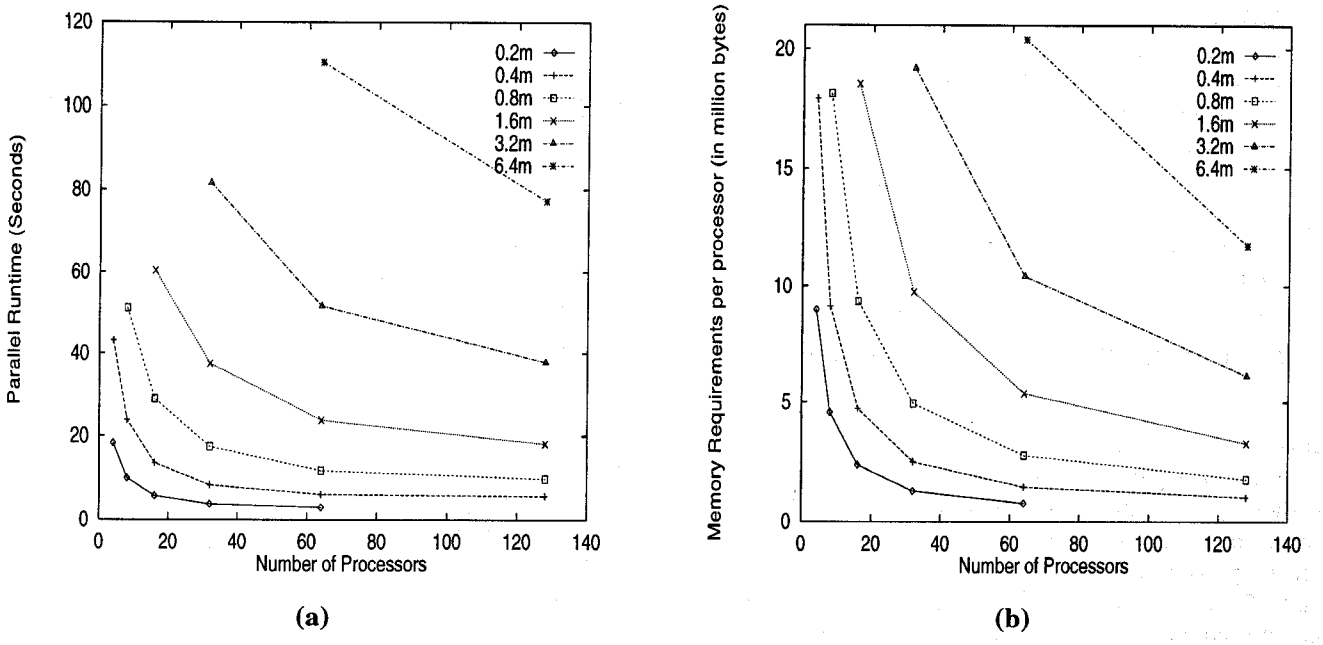

Figure 3. ScalParC behavior (a) Scalability for Parallel Runtime. (b) Scalability for Memory Requirements.

Figure 3(b) demonstrates the memory scalability of ScalParC by plotting the memory required per processor against the number of processors for various training set sizes. For smaller number of processors, the memory required drops by almost a perfect factor of two when the number of processors is doubled. Sizes of some of the buffers required for the collective communication operations increase with the increasing number of processors. Hence, for larger number of processors, we see a deviation from the ideal trend. In particular, for 0.8 million records, the memory required drops by a factor of 1.94 going from 8 to 16 processors, and it drops by a factor of 1.78 going from 32 to 64 processors.

Refer to [5] for more detailed results.

References

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

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

[3] P. K. Chan and S. Stolfo. Meta-learning for multistrategy and parallel learning. In *Proc. of Second International Workshop on Multistrategy Learning*, pages 150–165, 1993.

[4] D. J. Fifield. *Distributed tree construction from large datasets.* Bachelor's Honors thesis, Australian National University, 1992.

[5] M. V. Joshi, G. Karypis, and V. Kumar. Design of scalable parallel classification algorithms for mining large datasets. Technical Report 98-004, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1998. Also available at URL: http://www.cs.umn.edu/~kumar.

[6] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Algorithm Design and Analysis.* Benjamin-Cummings/Addison Wesley, Redwood City, CA, 1994.

[7] M. Mehta, R. Agarwal, and J. Rissanen. SLIQ: A fast scalable classifier for data mining. In *Proc. of 5th International Conference on Extending Database Technology (EBDT)*, Avignon, France, March 1996.

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

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

[10] J. Shafer, R. Agarwal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proc. of 22nd International Conference on Very Large Databases*, Mumbai, India, September 1996.