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

Mahesh V. Joshi† George Karypis† Vipin Kumar†

Abstract

In this paper, we present ScalParC (*Scal*able *Par*allel *C*lassifier), 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.

Keywords: Parallel Data Mining, Decision Tree, Classification.

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 because they are relatively inexpensive to construct, easy to interpret, and easy to integrate with the commercial database systems. For a variety of problem domains, they

*This work was supported by NSF CCR-9423082, by NSF ASC-9634719, 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.

†Department of Computer Science, University of Minnesota, Minneapolis, MN 55455

yield comparable or better accuracy as compared to other models such as neural networks, statistical models or genetic models [9]. The decision-tree based classifiers that handle large datasets are attractive, because use of larger datasets improves the classification accuracy even further[4]. Recently proposed classifiers SLIQ [2] and SPRINT [1] 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 [4, 5].

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 [10] and C4.5 [11] 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. A separate list is kept for each of the attributes, which maintains a record identifier for each sorted value. In the splitting phase, parts of a particular record from different lists need to be assigned to the same node of the decision tree. Implementation of this offers the design challenge. SPRINT builds a mapping between a record identifier and the node to which it goes to based on the splitting decision. The mapping is implemented as a hash table and is probed to split the attribute lists in a consistent manner. These hash tables are built on-the-fly for every node of the decision tree, and their size is proportional to the number of records at the node. For the upper levels of the tree, this size is of the same order as the size of 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 for development of parallel formulations for decision tree based classifiers [1, 7, 6]. Among these, the most relevant one is the parallel formulation of SPRINT[1], 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 total communication overhead per processor is $O(N)$, where N is the number of records in the dataset. Apart from the beginning phase of sorting, the serial runtime of a classifier is $O(N)$. Hence, SPRINT is unscalable in runtime.

It is unscalable in memory requirements also, because the total 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 and finally, some possible optimizations. Section 4 gives a detailed description of the algorithm and the data structures it uses. The experimental results are presented in Section 5. Section 6 contains concluding remarks.

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 [10] 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

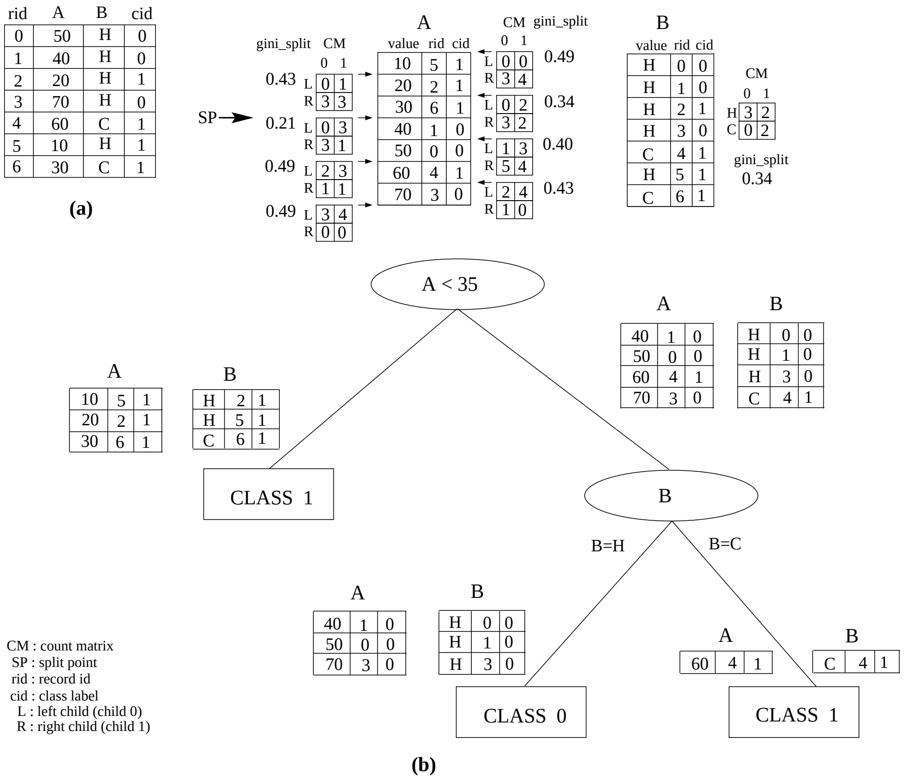


Figure 1: (a). An example training set. (b). Classification of the training set into a decision tree model.

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*[1]. 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[2], 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 the continuous attributes, 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

[1]Some classifiers allow a condition on a linear combination of many attributes [8], but we restrict ourselves to a single splitting attribute.

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

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*. Sorting the lists of continuous attributes on values introduces a relatively random order on record ids. This makes it necessary to associate a record id with each value in all the attribute lists.

The process described so far can be better understood using an example. Consider a sample training set shown in Figure 1(a). The class column is identified by *cid*. There are two attributes: a continuous attribute, A and a categorical attribute B. Attribute B can take two distinct values H and C. There are two possible class labels, 0 and 1. Figure 1(b) shows the entire tree induction process. First the training set is fragmented into attribute lists for A and B. They are shown after sorting A on values. The split determining process is illustrated in detail for the top node of the tree. A linear search made for the optimal value of A, is shown by the count matrices and their corresponding computed $gini_{split}$ values for all the possible positions. The decision $A < 35$ is found[3] to yield an optimal value of 0.21. If B were to serve as the splitting attribute, then it forms two partitions, one containing records with values H and other with C. The count matrix for this partitioning is shown in the figure along with the corresponding $gini_{split}$, which is 0.34. Hence, the overall least or optimal $gini_{split}$ is given by the decision $A < 35$, which is chosen to split the top node.

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. For example, in Figure 1, the splitting decision at the top node and the list of splitting attribute A yield the information that record ids $\{5,2,6\}$ go to child 0 (or the left child) and record ids $\{1,0,4,3\}$ go to child 1 (or the right child). 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 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 of splitting proposed until now, 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. This is shown in Figure 1, where the lists for attribute A at both the nodes of second level are sorted and have the class information needed to determine the split point. Once the splitting decision is made, it is

[3]The value used in the splitting condition is normally considered to be the average of two values around the split point.

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.

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 $O(T_s)$[3]; i.e., the parallelization overhead per processor should not exceed $O(T_s/p)$. This overhead, in general, consists of communication time and the overhead due to load imbalance. The algorithm design should try to minimize both by deciding on how the data is distributed among processors and how it is accessed. The design should also take into account the characteristics of the underlying parallel machine such as the latency and the bandwidth of the communication subsystem. Similarly, for scalability in memory requirements, the amount of memory required per processor should be $O(M/p)$, where M is the memory required on serial machine.

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 [3]. 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 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 $O(N/p)$.

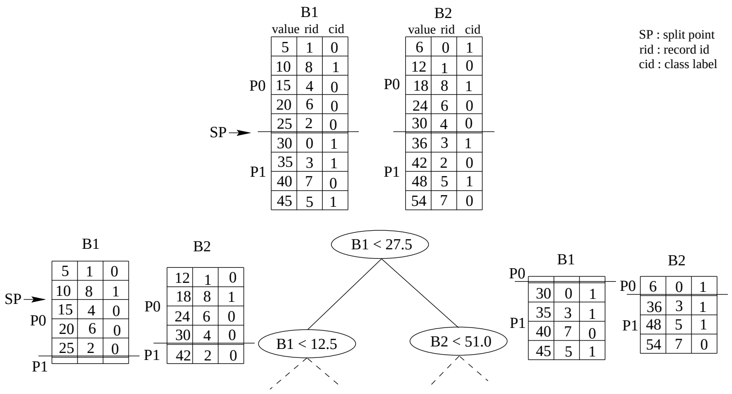


Figure 2: Demonstration of load imbalance of per-node approach and load balancing of per-level approach.

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 [7, 1].

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. Refer to Figure 2 for an example of such an imbalance. The root node in the figure has the best possible load balance, whereas the splitting decision makes both its children nodes to have a severe load imbalance. For the left child, processor $P0$ gets 9 entries to process (5 of $B1$ and 4 of $B2$) whereas processor $P1$ gets only one. Similar load imbalance can be observed for the other child. But, at the same time, if we look at the total number of records that are available per processor at the entire level, the balance among the processors is perfect. Hence, if the computations for *all* the records 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. Hence, for the purposes of achieving load balance and avoiding high latency costs, we choose to build the decision tree in breadth-first order and perform communications in a single phase at each level.

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 [1] 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 [1, 7].

The parallelization of the splitting phase is more difficult. In the parallel formulation of SPRINT, the hash table required for consistent splitting is built on *all* the processors for each node of the decision tree. Then each processor splits it local copies of all the attribute lists as in the sequential algorithm. Since each processor has to receive the entire hash table, the total 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, even if the communication is done on a per-node basis. This is because the size of the mapping needed per processor is proportional to the total number of records at that node, which is again $O(N)$ for the top node as well as for nodes at the upper levels of the tree.

3.3 The ScalParC Design 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 [3] 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 $\Omega(p)$ [3]. 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 3(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 process of constructing a distributed hash table. Figure 3(c) illustrates this process at the top node of the decision tree being built using the training set of Figure 3(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 3(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 3(d) for the enquiry process of *Age*.

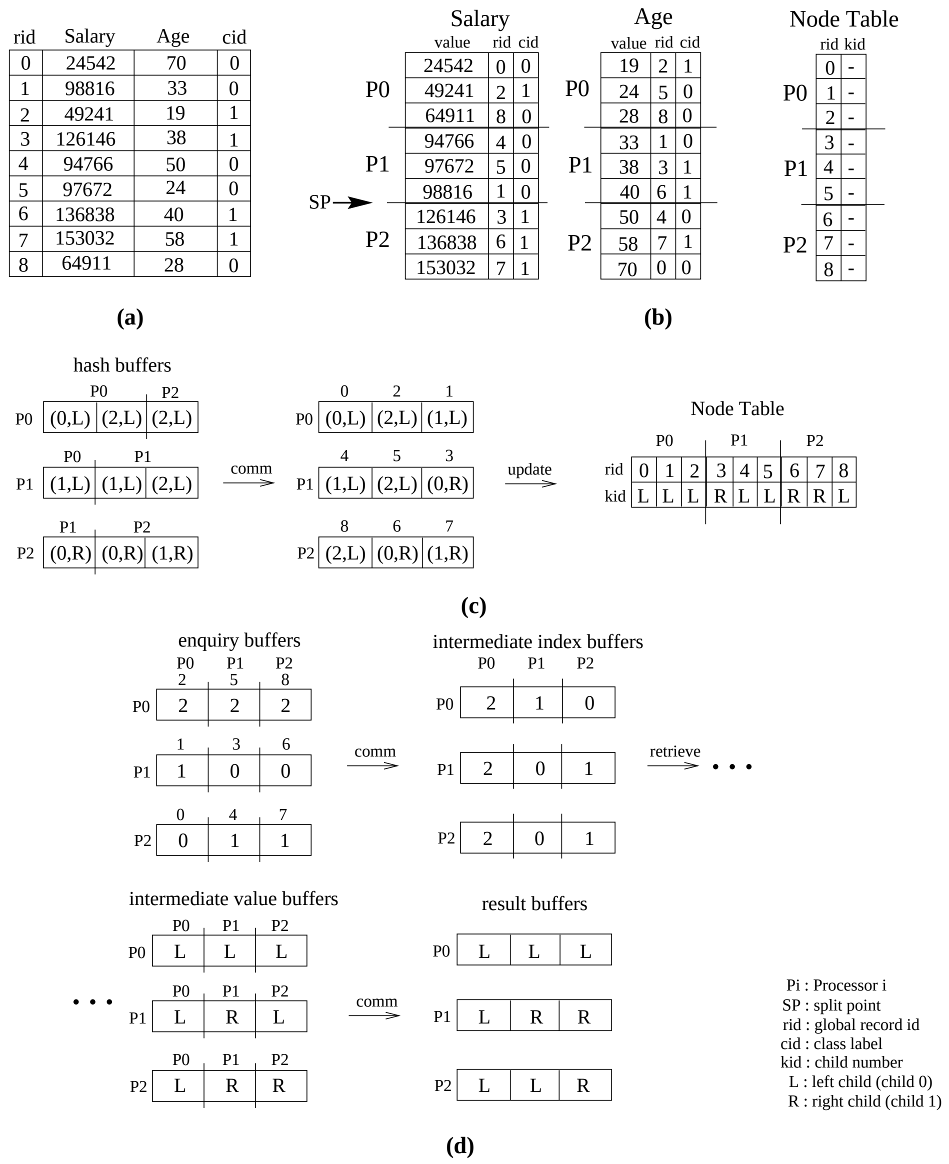


Figure 3: Illustrating the concept of distributed node table and the application of parallel hashing paradigm to splitting phase of ScalParC. (a) An example training set. (b) The attribute lists and empty node table at the beginning of splitting phase at the root node. (c) The process of hashing values 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.

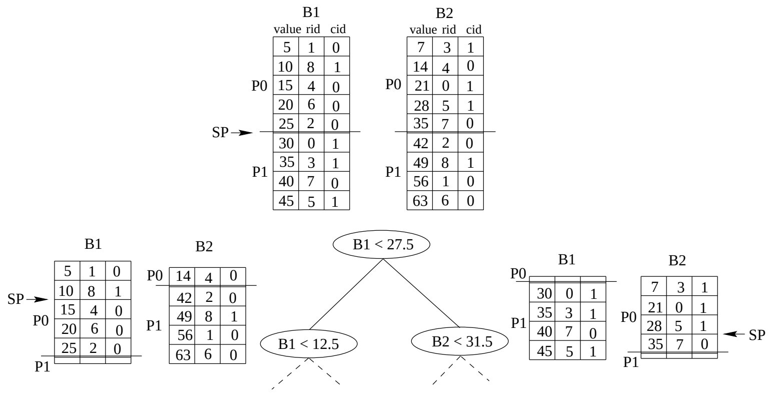


Figure 4: Depicts a scenario in which one processor might require to send $O(N)$ data while updating the node table using the per-level approach of communication.

3.3.3 Verifying Runtime and Memory Scalability

We now verify whether the application of the paradigm yields the desired scalability.

First consider the process of updating the node table. Note that, the communication is done on a per-level basis. So, each processor scans through its local lists of the splitting attributes for all the nodes at the current level of the decision tree in order to form the hash buffers. If the number of records at the current level of the tree is N_l, then no more than N_l entries in the node table will be updated at that level. Since $N_l = O(N)$ for the upper levels of tree, the overall communication overhead in updating the node table is $O(N)$[4]. Considering on a per-processor basis, no processor receives more than $O(N/p)$ updates, because the node table size on each processor is $O(N/p)$. Hence, the per-processor memory requirement for the receiving buffers also does not exceed $O(N/p)$. However, the number of updates sent by a processor depends on the distribution (among different processors) of the splitting attributes selected for different nodes at the current level. Although in most of the cases, the updates being sent will be equitably distributed among processors, there might be some situations in which some processors end up sending more than $O(N/p)$ updates. In this case, the runtime will be dominated by the processor that has the highest number of updates to send, which could be $O(N)$ in the worst case. Figure 4 shows a possible worst-case scenario, in which processor $P0$ has to hash all global record ids at the second level. Although the formulation becomes runtime unscalable in such a scenario, it can be seen that *no* parallelization approach which sorts continuous attributes only once, can avoid this unscalability. Also, from our experience, because of the relative randomness among the attribute lists, on an average, any given processor will not send more than 5-10% extra elements than N/p. Hence, such scenarios are rare and do not impact runtime scalability. Even in the worst case, our formulation still ensures memory scalability by dividing the updates being sent into blocks of N/p, hence requiring only $O(N/p)$ memory for the hash buffers.

[4]Recall that parallel formulation of SPRINT requires $O(N)$ communication overhead per processor resulting in $O(N\ p)$ overall communication overhead

In the process of enquiring the node table, for each of the attributes, no processor has more than $O(N/p)$ record ids to search at any level of the tree. Hence, at most $(n_a - 1)\ N/p$ values need to be searched to split all the remaining attributes. Thus, the total amount of data communicated in this process is $O(n_a\ N/p)$ per processor at any level. Hence, the formulation is runtime scalable for the enquiry process. The memory required during the process does not exceed $O(N/p)$, because the attribute lists are processed one after another and no processor needs to fill in more than $O(N/p)$ entries in the enquiry buffers for any given attribute. Also, since the size of the node table on each processor is $O(N/p)$, the intermediate index buffers (and the intermediate value buffers) require at most $O(N/p)$ memory. Hence, the formulation is memory scalable in enquiry process. Note that the memory required in both the update and enquiry processes can be expressed as $O(N/p + p)$ because some of the buffers required in the collective communication operations such as all-to-all-personalized operation, are of size proportional to number of processors. For N larger than $\Omega(p^2)$, the memory required is $O(N/p)$.

To summarize, applying the parallel hashing paradigm makes ScalParC truly memory scalable. Furthermore, it is also runtime scalable except for the pathological case as discussed above.

3.3.4 Optimizing Communication Overheads

The communication overheads can be further optimized by avoiding any communication for the categorical attributes. It is possible because the categorical attributes do not need any specific order on the record ids and hence they can be distributed to processors using the same distribution criterion as used for the node table. So, the communication overhead comes down from $O(n_a\ N)$ to $O(n_t\ N)$.

Another improvement over the communication overhead is possible. The intermediate index buffers formed in the search process, are fixed for each attribute throughout the classification process. This is true because the distribution of attribute lists among processors is fixed. Hence the communication overhead of the first step of sending enquiry buffers can be incurred only once. But, this saving requires extra storage for the intermediate index buffers. The total memory requirement goes up by $O(n_t\ N/p)$, and is still acceptable for scalability.

To put everything in perspective, the use of distributed node table allows a design that is scalable in both runtime and memory requirements. The next section gives detailed description of the data structures used and sketches the ScalParC algorithm.

4 The ScalParC Algorithm

4.1 Data Structures

- **Attribute Lists**.

  As noted in the section 3.1, the training set is fragmented vertically into separate lists for all attributes. The record id information is attached to every element in all the

Sort Continuous Attributes. **(Pre-sort)**
do while (there are nodes requiring splitting at the current level)
  Compute count matrices. **(Find-Split I.)**
  Compute best gini index for nodes requiring split. **(Find-Split II.)**
  Partition splitting attributes and Update the Node Table. **(Perform-Split-I.)**
  Partition non-splitting attributes. **(Perform-Split-II.)**
end do

Figure 5: ScalParC tree induction algorithm.

attribute lists. The choice of including the class labels in each list needs to be evaluated. The class labels are needed to compute the gini index. There are two options. One is to include the class labels in each of the lists. The other is to include them only in the distributed node table. The communication overhead of the latter approach is more, because both the node information and the class labels need to be communicated. Hence, for minimizing the overheads, we choose to store class labels in each of the lists. Each attribute list is fragmented horizontally into p fragments such that the first $p-1$ fragments have the same size. Fragment i is assigned to processor i. The continuous attribute lists will later be sorted in parallel to achieve a sorted order on values. The categorical attribute lists are assumed sorted on record ids before distributing them. As noted in section 3.3.4, this helps in optimizing the communication overheads for such attributes, provided that the node table is also distributed identically.

- **Node Table**. The global node table stores the node information for each record id. The record ids are implied by the table indices. If the splitting decision splits a node into m children, then the children are numbered from 0 to $m-1$. Each record can belong to only one node of the decision tree at any given instance, and it can propagate to a node only through the parent of that node. Hence, we need to store only the child number relative to a parent, for any record id. The global node table is fragmented horizontally into p fragments such the first $p-1$ fragments have the same size as that of the fragments of categorical attributes. Fragment i is stored on processor i.

- **Count matrices**. The count matrix $[n_{ij}]$, as described in section 2, is stored for each of the attributes. The choice of breadth-first order and per-level communication requires storing the count matrices of all the nodes at a given level. This storage can be reused for the next level.

4.2 Algorithm

Given distributed attribute lists, the ScalParC tree induction algorithm is shown in Figure 5. Each phase of the algorithm is explained in detail below. The algorithm is illustrated on three processors, using an example training set shown in Figure 3(a).

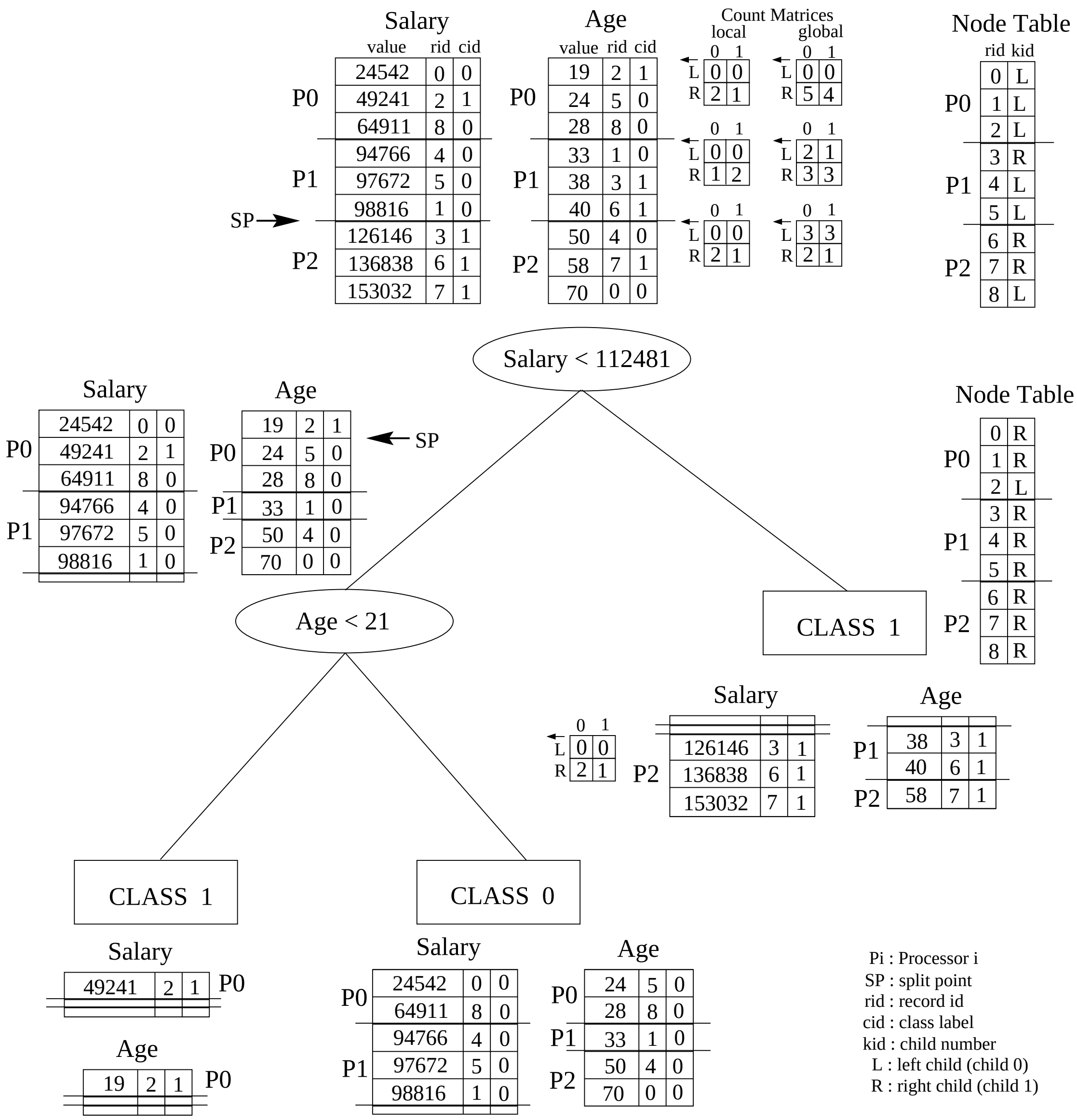


Figure 6: The ScalParC decision tree induction process.

- **Pre-sort.** We use the parallel sample sort algorithm followed by a parallel shift operation, to sort all the continuous attributes. This algorithm is scalable [3]. The shift operation is needed to maintain equal number of records on each processor initially and is performed very efficiently in our implementation. The result of the pre-sorting phase is shown in Figure 6 for both the continuous attributes: *Salary* and *Age*.

- **Find-Split-I.** For a continuous attribute, each processor first computes the local count matrix corresponding to the split position lying at the beginning of the local attribute list. Then a parallel prefix operation [3] is performed to compute the global count matrices for these split point positions. This process is illustrated in Figure 6, where for each processor, the local count matrix for attribute *Age* is shown for the beginning of the local part of the list. The figure also shows the global count matrices formed after the prefix operation. For a categorical attribute, there is only one count matrix possible per node. Each processor fills in its local copy of the count matrix. The global count matrix for that node is computed on a coordinating processor by doing a parallel reduction operation [3]. Note that, for any attribute, the count matrices from all the nodes at the current level of processing are combined before performing the collective communication.

- **Find-Split-II.** At first, for each node, the count matrix for one of the attributes is examined to determine whether the records at that node require to be split further. If all records bear one class label or alternately, the count matrix has only one non-zero entry, then that node does not need to be split further. Otherwise, for each continuous attribute, a processor finds the optimal split point (with minimum gini index) for its local part of the list. This is done by incrementally updating the count matrix and computing the gini index for each of the possible split point positions from the beginning to the end of the local list. For a categorical attribute, the assigned coordinator processor computes the gini index. The overall best splitting criterion is obtained by doing a parallel reduction operation. For example, Figure 6 shows the split points (SP) obtained after performing the reduction operation at both the decision nodes of the tree.

- **Perform-Split-I.** In this step, first the splitting attribute's list is split. For a continuous splitting attribute, each processor does this trivially by just updating the pointers into the list to demarcate the records belonging to each of the nodes at the next level. For a categorical splitting attribute, each processor rearranges its local list and updates pointers into this rearranged list. Secondly, each processor forms the hash buffers by scanning through the continuous splitting attribute lists of all the nodes at the current level. Then, the node table is updated using the process described in section 3.3.2. The updated node tables for the example-at-hand are shown in Figure 6 for each of the levels. As noted in section 3.3.3, there might be more than one communication steps needed to update the node table, in order to ensure memory scalability. The updates to the node table due to the categorical splitting attributes are applied locally and need not be communicated.

- **Perform-Split-II.** This step splits the lists for the non-splitting attributes, one attribute at a time. For a non-splitting continuous attribute, the node table is inquired using the enquiry process described in section 3.3.2. For a non-splitting categorical attribute, there is no communication required, as the required node information is available locally. The collected node information is then used to rearrange the attribute lists such that all the records for a given node are contiguous and a pointer is kept into this rearranged list for each new node. Figure 6 shows the rearrangement made for the example-at-hand. For the sake of better understanding, it does not show the pointers into the rearranged lists, but shows the lists as being physically split among nodes.

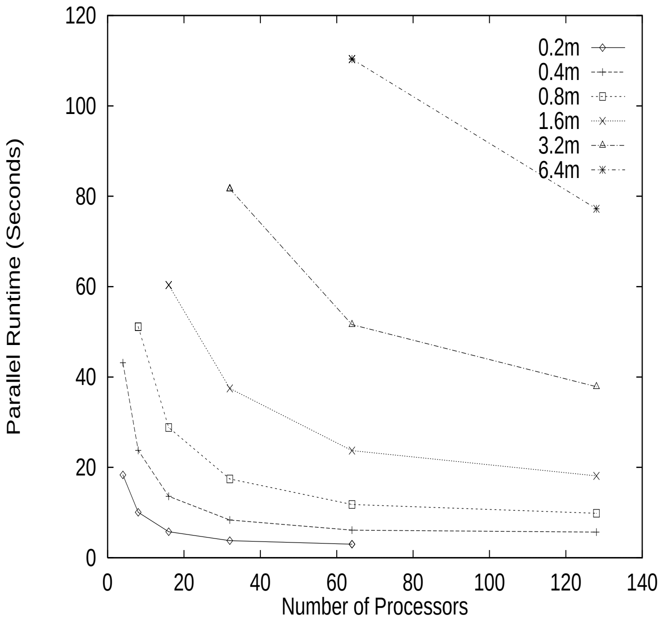
5 Experimental Results

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 for SPRINT[1].

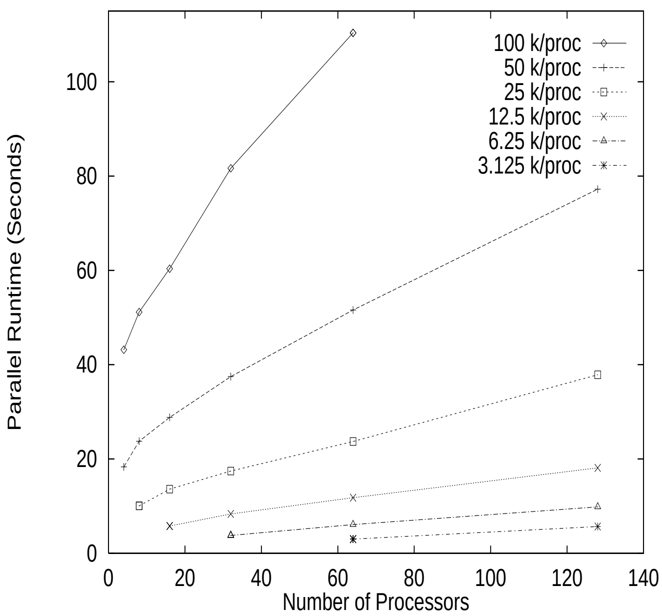
Figure 7(a) shows the runtime scalability of ScalParC by plotting the speedup obtained for various training set sizes. For a given problem instance, the relative speedups decrease as the number of processors are increased, because of increased overheads. In particular, for 1.6 million records, ScalParC 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[3]. 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.

Another look can be obtained at the parallelization overhead behavior using Figure 7(b). It plots the parallel runtime obtained on increasing number of processors while keeping the number of records per processor constant. The slope of a curve between two consecutive points is proportional to the relative speedup obtained. The slope increases with increasing number of records per processor, because the computation to communication ratio increases. For a given number of records per processor, the slopes decrease when larger number of processors are used, because of relative increase in the communication overheads.

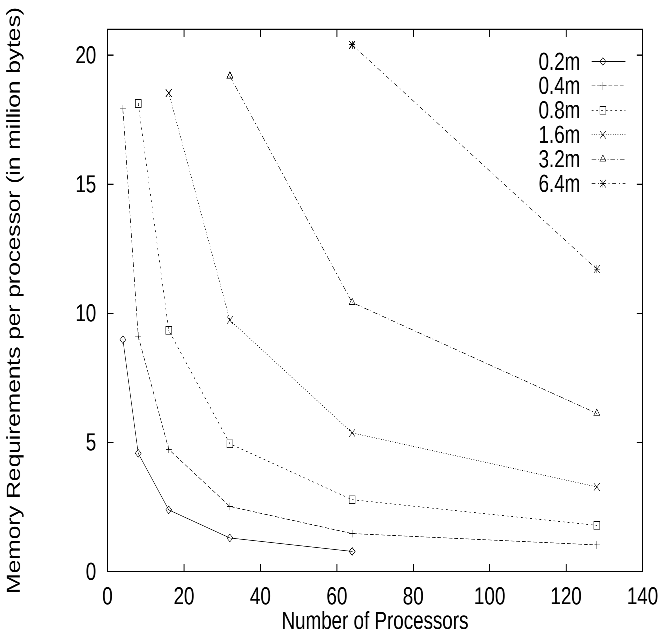
Figure 7(c) demonstrates the memory scalability of ScalParC by plotting the memory required per processor against the number of processors for various training set sizes. For smaller number of processors, the memory required drops by almost a perfect factor of two



(a)



(b)



(c)

Figure 7: ScalParC behavior (a) Scalability for Parallel Runtime in terms of speedup characteristics. (b) Scaleup Characteristics. (c) Scalability for Memory Requirements.

when the number of processors is doubled. As noted in section 3.3.3, 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. But, the rate of this drop decreases with the increasing training set size. In particular, going from 64 to 128 processors, the memory required drops by a factor of 1.74 for 6.4 million records, and it drops by a factor of 1.55 for 0.8 million records.

We observed the times taken by the Pre-Sort phase of ScalParC. In particular, time required to perform parallel sample sorting of one attribute list containing 100,000 records per processor on 64 processors was just around 3 seconds. This is very small compared to the corresponding classification time of 110 seconds. We also computed the load balance achieved for each of the four phases of the ScalParC tree induction algorithm and it was found that the maximum difference in the computation times of different processors did not exceed 10%.

We do not present the results for ScalParC with training sets including categorical attributes, because, by design, the scalability behavior of ScalParC remains unaffected with the inclusion of such attributes. The only communication overhead that is incurred for such attributes is that of communicating count matrices which form a very small fraction of the total runtime. Moreover, the gini index computation for categorical attributes is relatively much smaller than that for continuous attributes, and comes at no extra communication overheads. Hence, the addition of such attributes will tend to increase the efficiency of ScalParC slightly.

6 Concluding Remarks

This paper presented a parallel formulation of classification algorithms which are based on sorting of continuous attributes. This algorithm, ScalParC, is scalable in both parallel runtime and memory requirements. A MPI-based implementation of ScalParC was tested and the experimental results confirmed the scalable and efficient behavior of ScalParC for a wide range of training set sizes and on a wide range of processors. In particular, ScalParC achieved a relative speedup of around 7.1 from 4 to 64 processors on a relatively small problem of classifying 0.4 million records. It could classify 6.4 million records in just over a minute on 128 processors. Its memory requirements came down from 18.5MB/processor to 3.3 MB/processor when 1.6 million records were classified on 128 processors instead of 16 processors.

It should be noted that throughout the execution of ScalParC, the initial distribution of the attribute lists is not changed. This may cause some imbalance if the nodes that terminate in the upper levels of the tree have an uneven distribution of records among processors. However, an absolute load balance can be maintained by redistributing the lists among processors when the elimination of records start causing a severe load imbalance. This approach is explored in [7].

ScalParC's design allows it to handle very large datasets by using increasing number of

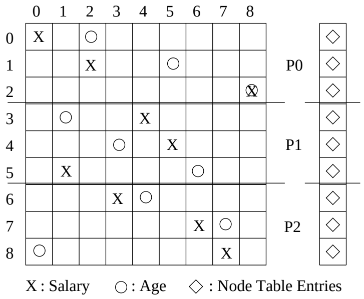


Figure 8: Similarity of ScalParC's communications to the Sparse Matrix Vector Multiplication algorithms.

processors. This is because, it requires just the local part of the distributed node table to remain in the main memory of each processor. All the attribute lists can reside on disks because they are always accessed linearly.

The key component of ScalParC is the parallel hash table. The parallel hashing paradigm can be used to parallelize other algorithms that require many concurrent updates to a large hash table. For example, it can be used to design an equally scalable parallelization of the SLIQ classifier [2], whose sequential approach is more efficient than that of SPRINT, but suffers a severe memory limitation. Refer to [12] for details of parallelizing SLIQ scalably.

The communication patterns used by ScalParC while doing the parallel hashing were inspired by those used in parallel sparse matrix-vector multiplication algorithms. Figure 8 demonstrates this similarity by transforming the splitting phase at the root node of Figure 6 into a sparse matrix-vector multiplication problem. The distributed node table acts as the the vector and the sparse matrix is formed by the record ids that need to be hashed. For example, as shown in Figure 8, the attribute list for *Salary* gets mapped into the non zeros marked by "X"s and the attribute list for *Age* gets mapped into the non zeros marked by "O"s. Note that the transformed sparse matrix will have at most n_t nonzero elements in each row, one for each of the continuous attributes. Some of the non zeros may overlap as shown in Figure 8. Well known parallel formulations of sparse matrix vector multiplication with horizontal striped partitioning of both the matrix and the vector formed this way, employ the same communication patterns as those of parallel hashing used in ScalParC.

References

[1] John Shafer, Rakesh Agrawal and Manish Mehta, *SPRINT: A Scalable Parallel Classifier for Data Mining*, Proc. of 22nd International Conference on Very Large Databases, Mumbai, India, Sept. 1996.

[2] Manish Mehta, Rakesh Agrawal and Jorma Rissanen, *SLIQ: A Fast Scalable Classifier for Data Mining*, Proc. of the 5th International Conference on Extending Database Technology (EDBT), Avignon, France, March 1996.

[3] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis, *Introduction to Parallel Computing: Algorithm Design and Analysis*, Benjamin-Cummings/Addison Wesley, Redwood City, 1994.

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

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

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

[7] Eui-Hong (Sam) Han, Anurag Srivastava, Vipin Kumar, *Parallel Formulations of Inductive Classification Learning Algorithm*, Technical Report 96-040, Department of Computer Science, University of Minnesota, Minneapolis, 1996.

[8] Sreerama K. Murthy, Simon Kasif and Steven Salzberg, *A System for Induction of Oblique Decision Trees*, Journal of Artificial Intelligence Research vol. 2, pp. 1-32, 1994.

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

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

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

[12] Mahesh V. Joshi, George Karypis and Vipin Kumar, *Design of scalable parallel classification algorithms via a new parallel hashing paradigm*, Technical Report under preparation, Department of Computer Science, University of Minnesota, Minneapolis, 1997.