Parallel Out-of-Core Divide-and-Conquer Techniques with Application to Classification Trees

Mahesh K. Sreenivas* smahesh@cise.ufl.edu
Khaled AlSabti† alsabti@ccis.ksu.edu.sa
Sanjay Ranka* ranka@cise.ufl.edu

Abstract

Classification is an important problem in the field of data mining. Construction of good classifiers is computationally intensive and offers plenty of scope for parallelization. Divide-and-conquer paradigm can be used to efficiently construct decision tree classifiers. We discuss in detail various techniques for parallel divide-and-conquer and extend these techniques to handle efficiently disk-resident data. Furthermore, a generic technique for parallel out-of-core divide-and-conquer problems is suggested. We present pCLOUDS, the parallel version of the decision tree classifier algorithm CLOUDS, capable of handling large out-of-core data sets. pCLOUDS exhibits excellent speedup, sizeup and scaleup properties which make it a competitive tool for data mining applications. We evaluate the performance of pCLOUDS for a range of synthetic data sets on the IBM-SP2.

1. Introduction

One of the important problems in the field of knowledge discovery and data mining is classification. It may be defined as the process of generating a description or a model for each class of a given data set. The data set consists of a number of example records, or examples in short, each consisting of one or more fields called *attributes* or *features*. Attributes can be *numeric* or *categorical*. Each record belongs to one of the given classes. The set of records available for developing classification methods may in general be decomposed into two disjoint subsets named *training set* and *test set*. The former is used for deriving the classifier while the latter is used for measuring the accuracy and generalization capabilities of the 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 [14]. Each non-leaf node of the tree contains a *splitter point* which is a test on one or more attributes and determines how the data is partitioned. The predicate—*until each partition consists entirely or dominantly of examples from one class*—defines the *stopping criteria*. Divide-and-conquer paradigm can be used to efficiently construct decision trees [7, 13, 14, 3].

The divide-and-conquer paradigm can be used for finding efficient solutions to a class of practical problems. These problems are recursive in nature. The execution of a problem instance is represented by a divide-and-conquer tree where each node represents a task or subtask.

Parallelization is inevitable for achieving substantial reduction in the overall response time while dealing with problems of large magnitude. *Task parallelism* and *data parallelism* are two basic techniques that may be used for the parallel construction of a divide-and-conquer tree. *Concatenated parallelism* is a variant of data parallelism, where multiple tasks are solved together using all the processors. Aluru et al. [4] present several applications of concatenated parallelism and show that this technique yields better results compared to data parallelism irrespective of the nature of the divide-and-conquer tree. However, the study does not address performance degradation due to I/O, normally associated with large out-of-core applications.

Srivastava et al. [16] present two approaches to parallel construction of trees: *synchronized tree construction* and *partitioned tree construction*. The former is based on the concept of concatenated parallelism and the latter employs task parallelism [4]. The synchronized tree construction gives good performance at upper levels of the tree unlike the partitioned tree construction approach which performs well at lower levels of the tree. Thus a hybrid approach is suggested by the authors for achieving better performance. The paper does not discuss the demanding I/O requirements of large out-of-core applications.

Parallel I/O is an expensive operation and can significantly affect the overall performance of parallel out-of-core algorithms [17]. This is especially true for dynamic and recursive problems where data needs to be fetched from the disk multiple number of times. A large body of work

* Department of CISE, University of Florida, Gainesville, FL 32611.
† A large part of this work was done while Khaled AlSabti was at University of Florida. His current address is: Department of CS, King Saud University, P.O. Box 51178, Riyadh 11543, Saudi Arabia.

has been done in the literature to study and design efficient parallel I/O algorithms [9, 17]. Kotz [9] advocates collective I/O for parallel file systems to realize high performance for I/O intensive applications. The study presents three implementation alternatives for collective I/O: traditional caching, two-phase I/O, and disk-directed I/O. The disk-directed model allows disk servers to control the order and timing of the flow of data for maximum performance. The data flow is thus controlled by the file system rather than by the application. Several useful paradigms for the design and implementation of efficient external memory algorithms (these algorithms are also known as out-of-core algorithms or I/O algorithms) can be found in Vitter [17]. This extensive survey concentrates on the I/O communication between the random access internal memory and the magnetic disk.

In this paper we present the parallel construction of an efficient and scalable decision tree classifier (pCLOUDS) used for mining large disk-resident data. pCLOUDS is the parallel version of a sequential decision tree classifier called CLOUDS [1, 3]. Before introducing pCLOUDS algorithm, we present a detailed discussion of various techniques for parallel divide-and-conquer paradigm. Further, these basic techniques are extended to include parallel I/O for efficient out-of-core parallel divide-and-conquer. We argue that data parallelism may be a better technique compared to concatenated parallelism for parallel external memory divide-and-conquer applications. Also, a generic technique for parallelizing out-of-core divide-and-conquer problems is suggested.

The rest of the paper is organized as follows. In Section 2 we present a brief discussion on coarse-grained parallel machines. Techniques for efficient parallel divide-and-conquer are presented in Section 3. We give a short discussion on various decision tree classifiers in Section 4. The parallel formulation of the classifier (pCLOUDS) is presented in Section 5. In Section 6, we present a performance evaluation of the pCLOUDS algorithm. We conclude with a summary in Section 7.

2. Coarse grained parallel machines

Coarse Grained Machines (CGMs) consist of a set of processors (tens to a few thousands) connected through an interconnection network. The main memory is physically distributed across the processors. Interaction between processors is either through message passing or through a shared address space.

In dealing with large out-of-core data sets, efficient management of disk-resident data is critical for minimizing performance degradation due to I/O. There are two main alternatives for handling the I/O subsystem: sequential and parallel. Sequential I/O gives poor performance for large I/O intensive applications and hence we focus only on parallel I/O in this paper. Further, we assume a shared-nothing type of architecture where each processor has its own disk which can be controlled independently. CGMs with cut-through routed networks is used for modeling the communication cost of various algorithms discussed in this paper.

Parallelization of applications requires distributing some or all of the data structures among the processors. Each processor needs to access all the non-local data required for its local computation. This generates aggregate or collective communication structures. Several algorithms have been described in the literature for these primitives and are part of standard textbooks [10].

The complexity analysis of the algorithms is presented for a p-processor hypercube interconnection architecture with cut-through routing. The analysis for permutation networks (e.g. IBM SP Series) and hypercubes is the same in most cases.

Primitive	**Time complexity**
All-to-all Broadcast	$O\left(\tau\ \log p\ +\ \mu\, m\,(p-1)\right)$
Gather	$O\left(\mu\ \log p\ +\ \tau\, mp\right)$
Global Combine	$O\left(\tau\ \log p\ +\ \mu\, m\right)$
Prefix Sum	$O\left(\tau\ \log p\ +\ \mu\, m\right)$

Table 1. Time complexity of primitives on a hypercube interconnection network

Table 1 describes the collective communication primitives, that may be used quite frequently in the discussions to follow, and their communication time requirements on cut-through routed hypercubes. Note that p refers to the number of processors. We model the cost of sending a message from one node to another as $O(\tau + \mu\, m)$, where m is the size of the message, τ represents the handshaking costs, and μ is the inverse bandwidth of the communication network. For more details see [2, 10, 15].

3. Parallel out-of-core divide-and-conquer techniques

Problem Statement Execution of a problem instance is represented by a divide-and-conquer tree. The root node contains the entire data set. Each internal node in the tree represents a task. The task is split into two subtasks[1] which correspond to the children of the node. The subtasks are solved recursively and the solution may have to be combined to find the solution for the parent task. For some problems, it may not be necessary to solve every subtask that is created at any given level in the tree. The cost of processing a subtask can be any function in the size of the subtask. For

[1] In this paper, we consider only binary trees since the extension of different techniques to n-ary trees is straightforward.

a problem instance of size n on p processors, we assume that the data is initially distributed at random among the p processors. Let the task of size n is divided into two subtasks of sizes n_l and n_r. We consider only those problems where $n_l + n_r \leq n$. The objective is to efficiently build the divide-and-conquer tree in parallel for the given problem. We assume that the entire data set cannot fully reside in the aggregate main memory of the parallel machine.

The initial distribution of the data points can affect the overall performance of different techniques for parallel divide-and-conquer. We use the theorem of Angluin and Valiant [5] for studying and designing the techniques and algorithms presented in this paper.

Theorem 1 *For a random distribution of n elements into b buckets, where $n \gg b$, the maximum number of elements of in a bucket is less than $\frac{n}{b} + O(c\sqrt{n/b}\ \log n)$ with high probability of $1 - n^{-c/2}$, where c is a positive constant.*

As a result of Theorem 1, the maximum number of elements assigned to each processor is less than $\frac{n}{p} + O(c\sqrt{n/p}\ \log n)$ with high probability of $1 - n^{-c/2}$. For large applications, it is expected that each processor to have almost n/p elements. Furthermore, we use Theorem 1 to prove the following lemma.

Lemma 2 *Let n elements be randomly distributed into b buckets, where $n \gg b$. Then, for any subset of size m elements, where $m \gg b$, the maximum number of elements in a bucket from the m elements is less than $\frac{m}{b} + O(c\sqrt{m/b}\ \log m)$ with high probability of $1 - m^{-c/2}$, where c is a positive constant. See [15] for proof.*

Several techniques for parallel in-core divide-and-conquer have been proposed in the literature. The popular ones are task parallelism, data parallelism, concatenated parallelism and mixed parallelism. We describe and compare each of these techniques and extend them for large out-of-core applications.

3.1. Task parallelism

In task parallelism processors are divided into subgroups and subtasks are assigned to processor subgroups based on the cost of processing each subtask. Subtasks belonging to different processor groups are solved concurrently.

There are two main approaches to handle the disk-resident data corresponding to the subtasks while applying task parallelism. In the first approach, when a subtask is assigned to a processor group, its data is also moved to that processor group. We call this I/O subsystem strategy as *compute dependent parallel I/O*. The data movement may be required for every internal node of the tree. This redistribution of data is very expensive operation; it requires the reading of data at source processors, communication to a destination processor group, followed by a writing step. For divide-and-conquer problems with linear (or near-linear) cost for computation, the redistribution cost may dominate the overall cost. At upper levels of the tree, the size of data to be processed at each node is relatively large and hence redistribution of data involves significant communication overhead in addition to the I/O. However, task parallelism involves no further communication overhead once the number of subtasks becomes equal to or greater than the number of processors. Task parallelism with compute dependent parallel I/O can provide very good performance when used at lower levels of the tree where the task grain size is small enough for in-core processing. The load balance can be improved with the presence of large number of such nodes.

When subtasks are assigned to processor subgroups it is not necessary to move the data to the destination processor subgroup. In this alternative approach we continue with the initial random distribution of data among the processors. Since data is not moved during processor regrouping we name this strategy as *compute independent parallel I/O*. For more details see [15].

3.2. Data parallelism

Tasks are solved in parallel using all the processors in data parallelism. This requires no movement of disk-resident data. Note that tasks are solved one after another in data parallelism. Solving a task using all the processors may require additional communication overhead.

Data parallelism may suffer from severe load imbalance because the subtasks may not be uniformly spread across the processors even if we ensure a uniform distribution of the parent task. Expensive redistribution of data may be needed to restore load balance in such a case. However, by Lemma 2, it is expected that each processor has approximately n/p of elements for a node of size n, where $n \gg p$. Thus, data parallelism is expected to achieve a good load balance for such cases. Data parallelism is very attractive for large out-of-core applications since it does not require any I/O overhead for data redistribution and the amount of I/O performed is balanced among the processors. For tasks (nodes) with large data, the cost of the synchronization and and message startups is small compared to the overall cost.

3.3. Concatenated parallelism

Concatenated parallelism is a variant of data parallelism where multiple tasks are solved collectively using all the processors [4]. Communication for all the subtasks can now be spooled together to save message startup costs for individual communication. Concatenated parallelism achieves good load balance even if individual subtasks are not dis-

tributed uniformly across the processors but their aggregate distribution is uniform.

In concatenated parallelism the available memory has to be shared by the many tasks that are solved together. This may lead to substantial I/O overhead for large out-of-core applications. With data distributed randomly across all the processors, data parallelism and concatenated parallelism are expected to provide similar load balance. Therefore, data parallelism seems to be a better alternative than concatenated parallelism for large external memory algorithms.

3.4. Mixed parallelism

For parallel out-of-core divide-and-conquer, better performance can be achieved by the use of mixed parallelism, since it allows us to combine the good aspects of basic techniques. There are different variants of mixed parallelism, choice of each one depends on the nature of the problem [15]. We primarily focus on an approach where data parallelism is used at upper levels of the tree. At lower levels of the tree we switch to task parallelism with compute dependent parallel I/O for managing disk-resident data.

In mixed parallelism, a criterion is needed for switching from one technique to another. For example, when the cost of task subdivision exceeds the cost of task redistribution we may stop using data parallelism and do redistribution for task parallelism. This switching criteria is generally problem dependent. The dynamic nature of the underlying I/O and computational pattern of divide-and-conquer problems makes the derivation of an optimal switching criteria very challenging.

For the problem studied in this paper, we use mixed parallelism in the following way. Nodes whose sizes are larger than a specified threshold are called *large* nodes and those which fall below the threshold are named *small* nodes. We use data parallelism for processing large nodes. It is recommended that the size of the large nodes be significantly larger than the number of processors. As the tree grows the number of examples at each node may decrease and as a consequence, computation time also decreases. However, there may not be a similar decrease in communication time since the message startup time is independent of the volume of communication. Hence communication time is expected to dominate the overall processing time when the node size becomes small. To overcome this, we switch to task parallelism for small nodes. Each small node is assigned to only one processor. The task-processor assignment is based on the task costs. The data of each task is redistributed to its destination processor by which building the subtree is performed locally within the processor. The assigning and processing of small nodes are delayed until all the large nodes have been processed to reduce the number of message startups. In this form of mixed parallelism data parallelism is followed by *delayed* task parallelism. Thus, the tree can be built in an arbitrary order. We will use the above technique for efficiently building decision tree classifiers.

4. Decision tree classifiers for data mining

Derivation of a decision tree classifier typically consists of a *construction* phase and a *pruning* phase. In construction phase we build the initial decision tree using the training data set. Pruning improves the generalization capabilities of the classifier by removing any over fitting to the training data set. We use an algorithm based on the *minimum description length* (MDL) principle to prune the decision tree [15]. The pruning phase can generally be executed in-memory and its cost is very small compared to that of the construction phase. We therefore focus only on parallelizing the construction phase.

The well-known decision tree classifiers CART [7] and C4.5 [13] build trees in a depth-first manner. The data is sorted repeatedly at every node of the tree to determine the best splits for numeric attributes. SLIQ [11] replaces this repeated sorting with one-time sorting by maintaining separate lists for each attribute. However, SLIQ uses a memory-resident data structure called *class list* which limits the number of input records it can handle. Another popular decision tree classifier that achieves good accuracy, compactness and efficiency for very large data sets is SPRINT [14].

In order to derive the splitting criterion at every internal node of the decision tree, classifiers like CART, SLIQ, and SPRINT, use the *gini* index. The use of gini index to derive the splitting criterion is computationally challenging. SPRINT uses a pre-sorting technique to calculate gini index for numeric attributes. However, the use of memory-resident hash tables to split the sorted attribute lists limits its scalability. ScalParc [8] is a more scalable parallel implementation of SPRINT algorithm.

CLOUDS (CLOUDS stands for Classification of Large Out of Core Data Sets) algorithm also uses the gini index to derive the splitting criterion [3]. CLOUDS mainly differs from SPRINT in the way it handles the numeric attributes. It presents two new methods for deriving the splitting point: *sampling the splitting points* (SS) and *sampling the splitting points with estimation* (SSE). Both methods evaluate only a subset of the entire splitting points along each numeric attribute. CLOUDS has substantially lower I/O and computational requirements compared to any other state-of-the-art classifiers. Moreover, the accuracy and compactness of the decision trees generated by CLOUDS remain the same or comparable to those of SPRINT.

4.1. Deriving the splitting point

CLOUDS evaluates categorical and numeric attributes differently. Categorical attributes are evaluated in the same way as in SPRINT. Therefore, we describe only the processing of numeric attributes.

4.1.1 Evaluation of numeric attributes

CLOUDS uses either the SS method or the SSE method to derive the splitter at each node of the tree. In the SS method the range of each numeric attribute is divided into q intervals such that each interval contains approximately the same number of points. These intervals are generated using a predrawn random sample set S. Gini indices are evaluated at the interval boundaries. These gini values, together with those computed for categorical attributes, are used to determine the minimum gini ($gini_{min}$). The splitting point with $gini_{min}$ is used as the splitter for the node under consideration. Note that this requires only one pass over the data set to derive the splitting point.

The SSE method goes beyond the SS method further as it estimates a lower bound for the gini value ($gini^{est}$) of each interval. The subset of intervals satisfying the inequality $gini^{est} < gini_{min}$ form a list of candidate intervals and are called *alive* intervals. For an alive interval we evaluate gini index at every distinct point in the interval to determine the best splitter. This may require another traversal through the entire data set. The SSE method effectively reduces the search space for the best splitting criterion. For more details on CLOUDS algorithm, the reader is referred to [1, 3, 15]. SSE method is more scalable and robust than SS method and hence we focus on the SSE method for the design of the parallel algorithm.

4.2. Partitioning the data set

The splitting criterion is applied to each record in the data and sample sets to determine their partition (left or right). The number of records read and written is equal to the number of records represented by the internal node. During the partitioning of the data set, the count matrices for categorical attributes, and the class frequencies at the interval boundaries for numeric attributes are also updated. This avoids a separate additional pass over the entire data to evaluate these frequencies.

5. Parallel formulation of CLOUDS

In this section we present the design and implementation of the pCLOUDS algorithm. The decision tree is built using mixed parallelism, with data parallelism for constructing the large nodes. For the small nodes, we use task parallelism, with compute dependent parallel I/O, where each small node is entirely assigned to one processor (cf. Section 3). The size of a small node is generally a few percent of the total data size.

For large applications, the cost of parallel I/O for divide-and-conquer problems can dominate the overall cost. In designing pCLOUDS, we gave special attention for the I/O. pCLOUDS maintains very good load balance for the performed I/O while keeping the associated overhead low for the overall process.

For small nodes, we use a direct method to arrive at the best split. In the direct method we sort the points along every numeric attribute and compute the gini index at each point. Further, these small nodes are processed in-memory. Large nodes are processed using data parallelism. For each large node of the tree, pCLOUDS performs the following steps:

1. Preprocessing
2. Deriving the splitting point
3. Partitioning of data and sample points

For a detailed description of each of the above steps see [15]. We discuss only those steps which are the most important. Note that all of the above steps, except portions of the preprocessing step, have to be performed at each large node of the tree and hence they collectively represent a task/subtask in the divide-and-conquer paradigm. In what follows, n denotes the total number of examples or records at some node of the decision tree. Therefore, at the root of the tree n denotes the total number of examples/records in the training set. Also, q denotes the number of intervals at some node. It should be noted that the value of q decreases as the node size decreases (as in CLOUDS).

5.1. Deriving the splitting point

pCLOUDS derives the splitting point by finding the point with the minimum (or near-minimum, as in CLOUDS) gini value. This is achieved by finding the minimum gini across all the categorical and numeric attributes. For details regarding the evaluation of categorical attributes see [15]. In the following subsections we briefly describe the sub-steps in evaluating the numeric attributes.

5.1.1 Evaluation of interval boundaries for numeric attributes

In this step the global frequency vectors of all the interval boundaries are generated, followed by evaluating the gini index at the boundary points to find the minimum gini among them. This step does not require any I/O. We consider two approaches for the implementation of this step. In the first method (the replication method), all the statistics of

the interval boundaries are replicated across all the processors. The second method (the distributed method) approximately distributes these statistics among the processors.

The replication method This method replicates the class frequency vectors for all boundary points at each processor for every numeric attribute. This requires $O(q\,c\,f)$ amount of storage at each processor, where c is the number of classes and f is the number of numeric attributes. The local vectors are combined to form the global vectors, followed by finding the minimum gini among the interval boundaries. This step can performed using one of the following three approaches.

- **Attribute-based approach** In this approach, all the global frequency vectors of each numeric attribute are assigned to only one processor. This requires $\delta\,O(q\,c) + O\,(\tau\ \log\ p\ +\ \mu\,q\,c)$ time for each numeric attribute. One of the main advantages of this option is that no further communication is needed for the calculation of gini index in the next step. However, during gini computation it is possible for some processors to idle and hence can lead to poor load balancing.

  In order to compute the gini index, we perform a prefix sum to update the frequency vectors at the boundary points. The prefix sum operation is completely local to the processor; it takes $\delta\ O(q\,c)$ time. Gini indices can, now, be calculated at all interval boundary points in $\delta\ O(q\,c)$ time. This approach has a total time complexity of $\delta\ O\,(q\,c)\ +\ O(\tau \log\ p\ +\ \mu\,q\,c)$ for each numeric attribute.
- **Interval-based approach** In this approach the global frequency vector of each interval is assigned to only one processor. In other words, the global frequency vectors for a numeric attribute are distributed across more than one processor. For more details see [15].
- **Hybrid approach** The third approach combines the above approaches to generate better load balancing. Details of this approach can be found in Sreenivas [15].

The global minimum gini (among the interval boundaries) is calculated by using a min-reduction primitive on the local minimum gini indices among the processors. $gini_{min}$ is thus the minimum value among the gini indices of categorical attributes and those at the interval boundary points of numeric attributes. We choose the attribute-based method to implement the replication method [15].

Distributed method Unlike in the replication method, the global frequency vectors are approximately distributed among the processors. This method can efficiently be implemented using efficient algorithms for *random access write* (RAW). For more details, the reader is referred to [6, 15].

The replication method is relatively simple to implement compared to the distributed method. Also, for large external memory applications, replication method incurs relatively less communication overhead compared to distributed method. For the above reasons, we choose the replication method for our implementation. A detailed comparison of these two methods is available in [15].

5.1.2 Determination of alive intervals

In this step, each processor determines the alive intervals (among the local intervals) using the sequential algorithm. This step does not require any I/O. An estimate ($gini^{est}$) for the minimum gini for each interval is computed (as in CLOUDS). The ratio of the number of examples in the alive intervals to the total number of examples is called *survival ratio*. The status of the intervals (alive or not) is broadcasted to all the processors for further evaluation of alive intervals. The communication cost of this step is proportional to $q\,c$.

5.1.3 Evaluation of alive intervals

For each alive interval, the gini index is calculated at all the potential splitting. During this process, each processor keeps track of the local minimum gini. The global minimum gini is found by using min-reduction on these local values. The corresponding splitting point is broadcasted to all the processors. This point is needed for partitioning the data. Evaluating the gini index for all the points in the alive intervals can be performed in several ways [15]. We briefly describe the single-assignment approach, which is used for our implementation. The communication cost of these approaches is proportional to the size of the alive intervals (we assume that the size of each alive interval is small enough to fit in the main memory).

Single-assignment approach In this case we assign each interval to only one (single) processor. One advantage of this approach is that no further communication is required after assigning the intervals to processors. The entire process can be performed locally. Assuming that the size of each alive interval is $O(n/q)$, it takes $\mu\ O(n/q)$ time to transfer the elements of each alive interval. The computation requirement for sorting the points in an alive intervals takes $\delta\ O\left(\frac{n}{q}\ \log\ \frac{n}{q}\right)$ time. Calculation of gini indices for an alive interval can be carried out in $\delta O(n\,c/q)$ time. Thus the total time complexity associated with this option for processing each alive interval is $\delta\ O\left(\frac{n\,c}{q}\ +\ \frac{n}{q}\ \log\ \frac{n}{q}\right)\ +\ \mu\,O\left(\frac{n}{q}\right)$. The assignment of intervals to processors is done based on the cost of processing each alive interval, i.e. the sorting cost.

5.2 Partitioning the data and sample points

As in the sequential case, once the splitting attribute and the splitting point are determined, the data and the sample points may be split accordingly. The cost of the I/O read and write, for each attribute, is proportional to $2n/p$. During partitioning, the local frequency vectors along all the numeric attributes as well as the count matrix for each categorical attribute for the left and right partitions are updated to avoid an additional pass over the entire data. The cost of the I/O, read and write, and computations across all the processors are (near) uniform. Further, this step does not require any communication, and gives almost perfect load balance.

pCLOUDS is expected to achieve a high degree of load balance for processing the large nodes. Further, the overall overhead for achieving load balance is small compared to the overall processing cost. This is due to keeping the I/O local to each processor as well as keeping the amount of I/O performed at each processor to be uniform [15].

6. Results and discussions

pCLOUDS is implemented using standard MPI communication primitives [12] and is relatively portable to different parallel architectures. Experiments were conducted on a 16-node IBM-SP2 shared nothing parallel processing supercomputer [15].

We use the data generator proposed in [11] to generate large synthetic data sets required for our experiments. A record in the data set consists of three categorical and six numeric attributes and a field describing the class label for the record. Each synthetic data set contains two classes and the records are generated by using the second classification function [11]. In the rest of the section we present the speedup, sizeup and scaleup characteristics of pCLOUDS.

We have measured the performance of pCLOUDS using training data sets varying in size from 3.6 to 7.2 million tuples. The data is distributed equally to all the processors at random, prior to the beginning of computation. We used a value of 10,000 for the number of intervals at the root. Large nodes are processed out-of-core if the size of those nodes exceed a pre-specified memory limit. We have used a memory limit of 1 MB for 6.0 million tuples. The value of memory limit is linearly scaled based on the size for other data sets. For switching from data parallelism to task parallelism we used a value of ten (in terms of the number of intervals) for the threshold [15].

Speedup characteristics of pCLOUDS is presented in Figure 1. As can be seen from Figure 1, speedup performance improves with increase in the data size. Superlinear speedup is obtained in some cases on four processors. This

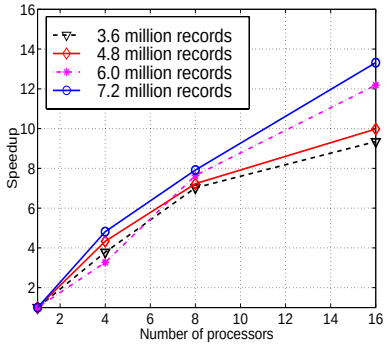

Figure 1. Speedup characteristics

is because of the cache effects, and the gain in I/O bandwidth with data being distributed across multiple disks. The speedup curve shows the presence of a glitch for 6.0 million records on four processors. Noticeable load imbalance is observed for this particular case in processing the alive intervals, possibly due to the presence of highly skewed intervals.

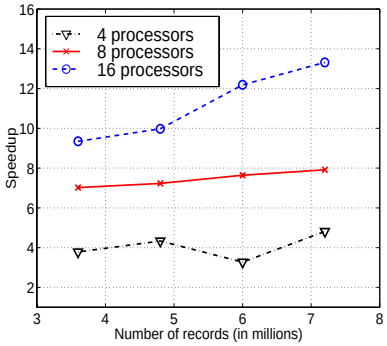

Figure 2. Sizeup characteristics

Figure 2 presents the performance of pCLOUDS for various data sizes. The gain in performance is marginal with increase in data size for the cases of four processors and eight processors. This is because the speedup is already close to the maximum even for 3.6 million tuples. However, for sixteen processors there is appreciable increase in speedup with increasing data size. As the data size increases the computation time also increases. On the contrary, the communication time required for exchanging count matrices and split points does not change much. This accounts for the superior sizeup performance of pCLOUDS.

The scaleup performance of pCLOUDS is presented in Figure 3. With each processor having the same amount of data, parallel runtime should ideally remain constant as the number of processors is increased. The experimental results show that pCLOUDS has excellent scaleup characteristics. The graphs show a near linear relationship between parallel runtime and the number of processors for different data densities. However, there is an increase in parallel runtime as the number of processors is increased. This is due to the

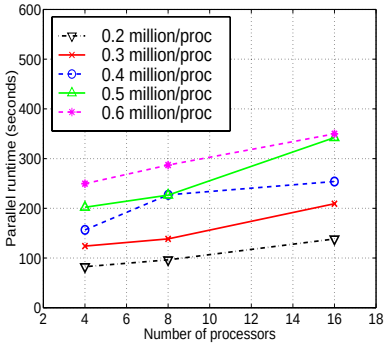

Figure 3. Scaleup characteristics

fact that we do not regroup the processors as they become idle, in our current implementation of task parallelism.

7. Conclusions

In this paper we have presented various techniques for the parallel construction of a generic divide-and-conquer tree. The basic techniques for parallelization are extended to efficiently handle I/O subsystem.

We have presented pCLOUDS, a parallel decision tree classifier for data mining. pCLOUDS is an application of out-of-core parallel divide-and-conquer paradigm. It uses the mixed approach for building the decision tree classifier. The performance evaluation and the complexity analysis show that pCLOUDS exhibits excellent speedup, scaleup and sizeup characteristics, and is hence a competitive tool for developing fast and accurate decision tree classifiers. The mixed approach, with data parallelism for large nodes, followed by delayed task parallelism for small nodes, is a very efficient technique for building divide-and-conquer trees for large applications.

We have not presented any concrete criteria for switching from data parallelism to task parallelism. This analytical characterization is currently under investigation.

Acknowledgment

We thank Prof. V. Kumar, P. Dokas and the Department of CS, University of Minnesota, for extending their parallel computing facility to conduct our experiments. We acknowledge the many fruitful discussions we had with H. Yoon. This research was supported in part by ARO under DAAG 55-97-1-0368 and Q000302 (subcontract from NMSU), and by NSF under CDA 96344470 and ASC 9318151 (subcontract 98031207 from Cornell University).

References

[1] K. AlSabti. *Efficient Algorithms for Data Mining*. Ph.D. Thesis, Syracuse University, 1998.

[2] K. AlSabti, S. Ranka, and R. Shankar. Many-to-many Personalized Communication with Bounded Traffic. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, February 1995.

[3] K. AlSabti, S. Ranka, and V. Singh. CLOUDS: A Decision Tree Classifier for Large Datasets. In *Proceedings of the International Conference on Knowledge Discovery in Databases and Data Mining*, New York, 1998.

[4] S. Aluru, S. Goil, and S. Ranka. Concatenated Parallelism: A Technique for Efficient Parallel Divide and Conquer. In *Proceedings of the International Conference on Parallel and Distributed Processing*, New Orleans, LA, 1996.

[5] D. Angluin and L. Valiant. Fast Probabilistic Algorithms for Hamiltonian Circuits and Matchings. *Journal of Computer Systems and Science*, 18(2):155–193, 1979.

[6] S. Bae. *Runtime Support for Unstructured Data Accesses on Coarse-Grained Distributed Memory Parallel Machines*. PhD thesis, Syracuse University, Syracuse, NY, 1997.

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

[8] M. V. Joshi, G. Karypis, and V. Kumar. ScalParC: A New Scalable and Efficient Parallel Classification Algorithm for Mining Large Datasets. In *Proceedings of the International Parallel Processing Symposium*, Orlando, FL, April 1998.

[9] D. Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 61–74, Monterey, CA, November 1994.

[10] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing*. The Benjamin/Cummings Publishing Company, Inc., CA, 1994.

[11] M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A Fast Scalable Classifier for Data Mining. In *Proceedings of the 5th International Conference on Extending Database Technology*, pages 18–32, Avignon, France, 1996.

[12] *MPI: A Message Passing Interface Standard*. Message Passing Interface Forum, May 1994.

[13] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufman, San Mateo, CA, 1993.

[14] J. C. Shafer, R. Agrawal, and M. Mehta. SPRINT: A Scalable Parallel Classifier for Data Mining. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, pages 544–555, Mumbai, India, 1996.

[15] M. K. Sreenivas. Parallel Out-of-Core Decision Tree Classifiers. Master's thesis, University of Florida, Gainesville, FL, December 1998.

[16] A. Srivastava, E.-H. Han, V. Kumar, and V. Singh. Parallel Formulations of Decision-Tree Classification Algorithms. In *Proceedings of the International Conference on Parallel Processing*, Minneapolis, MN, 1998.

[17] J. S. Vitter. External Memory Algorithms. In *Proceedings of the 17th Annual ACM Symposium on Principles of Database Systems (PODS '98)*, pages 119–128, Seattle, WA, June 1998.