On the Design of a Parallel Object-Oriented Data Mining Toolkit

Chandrika Kamath
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
P.O. Box 808, L-561
Livermore, CA 94551
kamath2@llnl.gov

Erick Cantú-Paz
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
P.O. Box 808, L-551
Livermore, CA 94551
cantupaz1@llnl.gov

ABSTRACT

As data mining techniques are applied to ever larger data sets, it is becoming clear that parallel processors will play an important role in reducing the turn-around time for data analysis. In this paper, we describe the design of a parallel object-oriented toolkit for mining scientific data sets. After a brief discussion of our design goals, we describe our overall system design that uses data mining to find useful information in raw data in an iterative and interactive manner. Using decision trees as an example, we illustrate how the need to support flexibility and extensibility can make the parallel implementation of our algorithms very challenging. We describe the solution approaches we are considering to address these challenges. As this is work in progress, we also present some preliminary results using an astronomy data set.

1. INTRODUCTION

Parallel data mining is the exploitation of fine grained parallelism in data mining, using tightly-coupled processors connected by a high-bandwidth interconnection network [8]. Implicit in this definition is the assumption that all the data used in mining is locally available, not globally distributed. This is often the case when commercial or scientific data is collected at one location, and often analyzed at the same location. If the size of the data is very large or a fast turn-around is required, it may be appropriate to mine the data using a parallel system. With 2-16 processor, Intel-based systems becoming inexpensive and common-place, the compute power necessary to implement this fine-grained parallelism is readily available.

Local data can be mined using either tightly- or loosely-coupled processors. In both cases, we need to focus on minimizing the communication costs across the processors. However, for loosely-coupled processors, this communication cost is typically much larger and may suggest the use of distributed data mining techniques, where the data is globally distributed, and communication done via the Internet.

In this paper, we discuss the issues involved in designing and implementing an object oriented framework for mining data using tightly-coupled processors. Our focus is on distributed memory architectures where each compute node has its own memory, and the nodes share only the interconnect. The architecture of such systems is scalable with increasing number of processors, making them well suited to mining massive data sets.

The outline of this paper is as follows: In Section 2, we describe our view of data mining in light of the challenges we face in mining scientific data. Next, we describe the system architecture we have designed for Sapphire, a large-scale data mining project at the Lawrence Livermore National Laboratory [14]. In Section 3, we use decision trees as an example to illustrate the problems we face, and the tradeoffs we have made, as our need for flexibility meets the realities of parallel implementation. In Section 4, we discuss our experiences in mining an astronomy data set. Section 5 describes our experimental results, and in Section 6, we conclude with a summary.

2. THE DATA MINING PROCESS

The tasks that are performed in a data mining application depend on the problem domain, the problem being solved, and the data itself. The Sapphire toolkit described in this paper is targeted to problems arising mainly from scientific applications, where the data is obtained from observations, experiments, or simulations. Scientific data analysis, while varied in scope, has several common challenges:

- **Feature extraction from low-level data:** Science data can be either image data from observations or experiments, or mesh data from computer simulations of complex phenomena, in two and three dimensions, involving several variables. This data is available in a raw form, with values at each pixel in an image, or each grid point in a mesh. As the patterns of interest are at a higher level, additional features must be extracted from the raw data prior to pattern recognition.

- **Noisy data:** Scientific data, especially data from observations and experiments, is noisy. This noise may vary within an image, from image to image, and from sensor to sensor. Removing the noise from data, without affecting the signal is a challenging problem in scientific data sets.
- **Size of the data:** Our data sets range from moderate to massive, with the smallest being measured in hundreds of Gigabytes and the largest a few Terabytes. As more complex simulations are performed, the data is expected to grow to the Petabyte range.
- **Need for data fusion:** Frequently, scientific data is collected from various sources, using different sensors. In order to use all available data in the analysis, we need data fusion techniques. This is a non-trivial task if the data was collected at different resolutions, using different wavelengths, or under different conditions.
- **Lack of labeled data:** Labeled examples in scientific data are usually generated manually. This tedious process is made more complicated as not all scientists may agree on a label for an object, or want the data mining algorithm to identify "interesting" objects, not just objects that are similar to the training set.
- **Data in flat files, not data bases:** Unlike commercial data, scientific data is rarely available in a cleaned state in data warehouses.
- **Mining data as it is being generated:** In the case of simulation data, scientists are interested in the behavior of the scientific phenomena as it changes with time. Sometimes, the time taken to output the result of the simulation at each time step can exceed the simulation time itself. Since the simulations are run on large parallel computers, it may be possible to preprocess the data while it is being generated, resulting in a smaller output. While this idea seems simple, a practical implementation is non-trivial.

In light of these conditions, our definition of data mining starts with the raw data and includes pre-processing prior to pattern recognition (Figure 1). If the raw data is very large, we may use sampling and work with fewer instances, or use multiresolution techniques and work with data at a coarser resolution. This first step may also include data fusion, if required. Next, noise is removed and relevant features are extracted from the data. This results in a feature vector for each data instance. Depending on the problem and the data, we may need to reduce the number of features using dimension reduction techniques such as principal component analysis (PCA). After this pre-processing, the data is ready for the detection of patterns. These patterns are then displayed to the user, who validates them appropriately.

The data mining process is iterative and interactive; any step may lead to a refinement of the previous steps. User feedback plays a critical role in the success of data mining in all stages, starting from the initial description of the data, the identification of potentially relevant features and the training set (where necessary), and the validation of the results.

2.1 The Sapphire System Design

In order to implement the data mining process in Figure 1 in a parallel setting, our experience has shown that we must take into account the following facts:

- Not all problems require the entire data mining process, so each of the steps must be modular and capable of stand-alone operation.
- Not all algorithms are suitable for a problem, so the software should include several algorithms for each task, and allow easy plug and play of these algorithms.
- Each algorithm typically depends on several parameters, so the software should allow user friendly access to these parameters.
- Intermediate data must be stored appropriately to support refinement of the data mining process.
- The domain dependent and independent parts must be clearly identified to allow maximum re-use of software as we move from one application to another.

To accommodate these requirements, we put together the system architecture shown in Figure 2. The focus of Sapphire is on the compute-intensive tasks as these benefit the most from parallelism. Such tasks include decision trees, neural networks, image processing, and dimension reduction. Each class of algorithms is designed using object-oriented principles and implemented as a C++ class library. Parallelism is supported through the use of MPI for distributed-memory parallel processing [9]. We use domain-specific software for tasks such as reading, writing, and the display of data. To support many different input data formats, such as FITS, View, and netCDF, we first convert each format into Sapphire's internal data format, prior to any processing. We are using RDB, a public-domain relational data base, as our permanent data store to store the intermediate data generated at each step. This has turned out to be invaluable as it has allowed us to experiment with different subsets of features and enabled us to easily support a growing data set. Our ultimate goal is that once we have each of the class libraries implemented, we will be able to provide a solution to a problem in a domain by simply linking the appropriate algorithms using a scripting language such as Python.

The system design of our parallel object-oriented toolkit has been made challenging by the following two factors:

- As data mining proceeds from feature extraction to the discovery of useful information, the data processed reduces in size. This reduction can be very drastic, e.g. from a Terabyte to a Megabyte. Furthur, some of the data pre-processing could occur on the parallel machine where the data is being generated, while the rest of the data analysis could take place on a different parallel machine with possibly fewer processors. Ensuring the end-to-end scalability of the data mining process under these circumstances could prove very challenging.

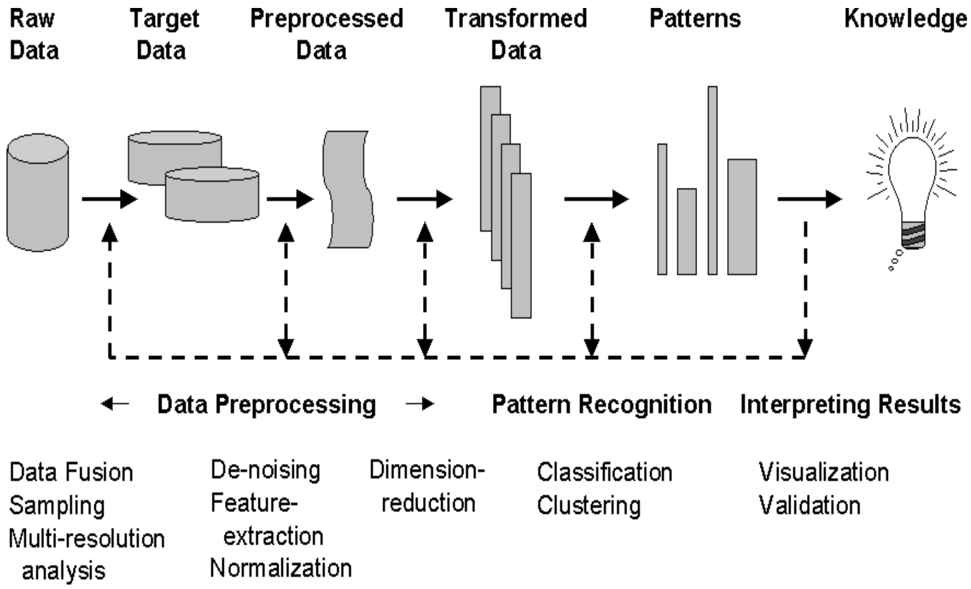


Figure 1: Data mining: an iterative and interactive process.

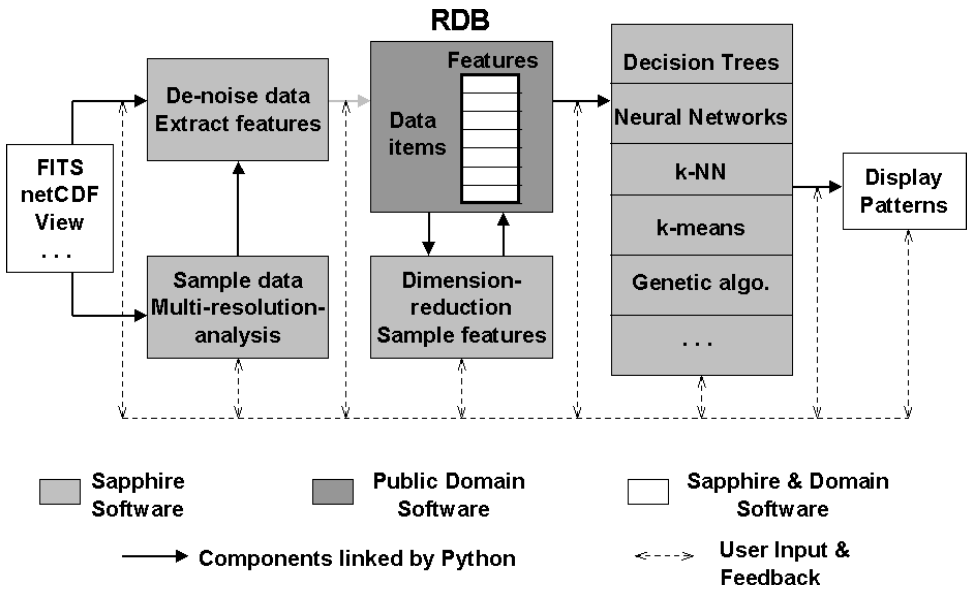


Figure 2: The Sapphire System Architecture: Flexible and Extensible

- The very nature of data mining requires close collaboration with the domain scientists at each step. Incorporating this iterative and interactive aspect into a parallel framework is a non-trivial task.

We next focus on one of the algorithms in data mining, namely, decision trees, describe our approach to the design and implementation of parallel software, and show how the need to support flexibility in a parallel implementation can give rise to conflicting requirements.

3. PARALLELIZING DECISION TREES

Decision trees [3, 12, 11] belong to the category of classification algorithms wherein the algorithm learns a function that maps a data item into one of several pre-defined classes. These algorithms typically have two phases. In the training phase, the algorithm is "trained" by presenting it with a set of examples with known classification. In the test phase, the model created in the training phase is tested to determine how well it classifies known examples. If the results meet expected accuracy, the model is put into operation to classify examples with unknown classification. This operation is embarrassingly parallel as several "copies" of the classifier can operate on different examples. It is important for the training phase of the classifier to be efficient as we need to find an optimum set of parameters which will enable accurate and efficient results during the operation of the classifier.

A decision tree is a structure that is either a leaf, indicating a class, or a decision node that specifies some test to be carried out on a feature (or a combination of features), with a branch and sub-tree for each possible outcome of the test. The decision at each node of the tree is made to reveal the structure in the data. Decision trees tend to be relatively simple to implement, yield results that can be interpreted, and have built-in dimension reduction. Parallel implementations of decision trees have been the subject of extensive research in the last few years [15, 17, 20, 16]. An approach used to construct a scalable decision tree was first described in the SPRINT algorithm [15]. Instead of sorting the features at each node of the tree as was done in earlier implementations, it uses a single sort on all the features at the beginning. The creation of the tree is thus split into two parts:

Initial Sorting:

- First the training set is split into separate feature lists for each feature. Each list contains the identification (ID) number of the data instance, the feature value, and the class associated with the instance. This data is partitioned uniformly among the processors.
- Next, a parallel sort is performed on each feature list which results in each processor containing a static, contiguous, sorted portion of the feature. As a result of this sort, the data instances for one feature in one processor may be different from the data instances for another feature in the same processor. Since all the features corresponding to one data instance may not belong to the same processor, it is important to include the ID number of the data instance in the feature list.
- Next, we build count statistics for each of the features in each processor.

Creation of the decision tree:

- Find the optimal split point:
  - Each processor evaluates each of the local feature lists to find the best local split (this is done in parallel by all processors).
  - It communicates the local best splits and count statistics to all processors.
  - Each processor determines the best global split (this is done in parallel by all processors).
- Split the data:
  - Each processor splits on the winning feature, and sends the ID numbers of its new left and right node data instances to all other processors.
  - Then, each processor builds a hash table containing all the ID numbers, and information on which instances belong to which decision tree node.
  - Next, each processor, for each feature, probes the hash table for each ID number to determine how to split that feature value.
- This process is carried out on the next unsolved decision tree node.

An improved version of the SPRINT algorithm that is scalable in both run-time and memory requirements is described in ScalParC [7]. This differs from SPRINT in two ways. First, a distributed hash table is used, instead of a single hash table which is replicated in each processor. This reduces memory requirements per processor, making the algorithm scalable with respect to memory. Second, as in SPRINT, the decision tree nodes are constructed breadth-first rather than depth-first and processor synchronization is held off until all work is done for that level of the tree. This not only limits the communication necessary for synchronization, but also results in better load balancing since processors that finish with one node of the tree can move directly on to the next node.

3.1 Decision Tree Functionality

Our goal in the design and implementation of the Sapphire decision tree software is to take the ScalParC approach and extend it to include support for different types of splits and different splitting criteria. We next describe the options we want to support in our software.

Support for several different splitting criteria:
The feature to test at each node of the tree, as well as the value against which to test it, can be determined using one of several measures. Depending on whether the measure evaluates the goodness or badness of a split, it can be either maximized or minimized. Let T be the set of n examples at a node that belong to one of k classes, and T_L and T_R be the two non-overlapping subsets that result from the split (that is, the left and right subsets). Let L_j and R_j be the number of instances of class j on the left and the right, respectively. Then, the split criteria we want to support include [10]:

- *Gini:* This criterion is based on finding the split that most reduces the node impurity, where the impurity is defined as follows:

$$L_{Gini} = 1.0 - \sum_{i=1}^{k} (L_i / |T_L|)^2$$

$$R_{Gini} = 1.0 - \sum_{i=1}^{k} (R_i / |T_R|)^2$$

$$\text{Impurity} = (|T_L| * L_{Gini} + |T_R| * R_{Gini})/n$$

where $|T_L|$ and $|T_R|$ are the number of examples, and L_{Gini} and R_{Gini} are the Gini indices on the left and right side of the split, respectively. This criterion can have problems when there are a large number of classes.

- *Twoing rule:* In this case, a "goodness" measure is evaluated as follows:

$$\text{Twoing value} = (|T_L|/n) * (|T_R|/n) * \left(\sum_{i=1}^{k} |L_i/|T_L| - R_i/|T_R|| \right)^2$$

Twoing may be a more desirable split criterion when there are a large number of classes. Comparisons between twoing and Gini indicate only slight differences between the two, though Breiman et al. [3] prefer Gini as the splits produced by it appear to be better in terms of producing pure descendant nodes.

- *Information gain:* The information gain associated with a feature is the expected reduction in entropy caused by partitioning the examples according to the feature. Here the entropy characterizes the (im)purity of an arbitrary collection of examples. For example, the entropy prior to the split in our example would be:

$$\text{Entropy}(T) = \sum_{i=1}^{k} -p_i \log_2 p_i$$

$$p_i = (L_i + R_i)/n$$

where p_i is the proportion of T belonging to class i and $(L_i + R_i)$ is the number of examples in class i in T. The information gain of a split $S = \{T_L, T_R\}$ relative to T is then given by

$$\begin{aligned} \text{Gain}(T,S) &= \text{Entropy}(T) \\ &- \frac{|T_L|}{|T|} * \text{Entropy}(T_L) \\ &- \frac{|T_R|}{|T|} * \text{Entropy}(T_R) \end{aligned}$$

where T_L and T_R are the subsets of T that correspond to the left and right branches, respectively. This criterion tends to favor features with many values over those with few values.

Support for non-axis-parallel decision trees: Traditional decision trees consider a single feature at each node, resulting in hyperplanes that are parallel to one of the axes. While such trees are easy to interpret, they may be complicated and inaccurate in the case where the data is best partitioned by an oblique hyperplane. In such instances, it may be appropriate to make a decision based on a linear combination of features, instead of a single feature. However, these oblique trees can be harder to interpret. They can also be more compute intensive as the problem of finding an oblique hyperplane is much harder than the problem of finding an axis-parallel one. None-the-less, our early research has shown that when used in conjunction with evolutionary algorithms, these oblique classifiers could prove competitive in some cases [5]. To further explore these ideas, we are designing our software such that, in addition to axis parallel trees, it can support the following types of splits at each node:

- *CART-LC:* Breiman et al., in [3], suggested the use of linear combinations of features to split the data at a node. If the features for a data instance are given as $(x_1, x_2, \ldots, x_n, c)$, where c is the class label associated with the instance, then, we search for a best split of the form

$$\sum_{i=1}^{n} a_i x_i \leqslant d \quad \text{where} \quad \sum_{i=1}^{n} a_i^2 = 1$$

and d ranges over all possible values. The solution approach cycles through the coefficients $a_1, \ldots, a_n$, trying to find the best split on each coefficient, while keeping the others constant. A backward deletion process is then used to remove coefficients that contribute little to the effectiveness of the split. This approach is fully deterministic and can get trapped in a local minima.

- *OC1:* The oblique classifier OC1 described in [10] attempts to address some of the limitations of the CART-LC approach by including randomization in the algorithm that finds the best hyperplane. Further, multiple random re-starts are used to escape local minima. In order to be at least as powerful as the axis-parallel decision trees, OC1 first finds the best axis-parallel split at a node before looking for an oblique split. The axis-parallel split is used if it is better than the best oblique split determined by the algorithm for that node. OC1 also shifts to an axis-parallel split when the number of examples at a node falls below a user-specified threshold to ensure that the data does not underfit the concept to be learned.

- *Oblique-EA:* In this approach, we use evolutionary algorithms to find the best hyperplane represented by the coefficients $(a_1, \ldots, a_n, d)$. An individual in the population is represented by the concatenated version of these coefficients. The fitness of each individual is determined by evaluating how well it splits the examples at a node for a given split criterion. Evolutionary algorithms thus allow us to work with all the coefficients at a time instead of the series of univariate splits considered in OC1 and CART-LC.

  We have explored two options for evolutionary algorithms. In one case we use a (1+1) evolutionary strategy with adaptive mutations. The initial hyperplane is the best axis-parallel split for the node. For each hyperplane coefficient, we have a mutation coefficient, which is updated at each iteration and used to determine the new hyperplane coefficient. We then select the best between the parent and child hyperplanes. In the second approach, we use a simple generational GA with real valued genes. The initial population consists of 10% copies of the axis-parallel hyperplane, and the rest are generated randomly. Our initial experiments have shown that in some cases, the Oblique-EA approaches are faster and more accurate than OC1 [5].

Our main challenge is to see if we can support these different options by isolating, and re-using, the code that supports the parallel implementation of each option.

We are also interested in exploring different ways to avoid over-fitting through pruning and rules that decide when to stop splitting, such as the cost complexity pruning technique of Breiman [3] or the minimum description length approach suggested by Quinlan and Rivest [13]. However, since pruning takes place after the creation of the tree, and is not computationally intensive to benefit from parallel processing, we do not address the topic in this paper.

3.2 The Sapphire Decision Tree Design

As explained in the previous section, we are interested in a decision tree design that gives us enough flexibility to experiment with different options within a parallel implementation. It is relatively easy to support some of these options within the context of an object-oriented design. For example, different splitting criteria can be easily supported by having an abstract base class from which concrete classes for the split criterion are inherited. These concrete classes implement the function used to determine the quality of a split. The user can then instantiate an object in one of these classes to indicate the split criterion used at all nodes of the tree. This choice would be communicated to the decision tree object by passing a pointer to the base split criteria class as an argument. A similar situation holds in the case of pruning options which are executed after the tree is built. In both cases, the main operation performed by the class is at a low-enough level that no parallelism is required in the implementation of the operation.

The UML class diagram [18] for our decision tree design is given in Figure 3. The prefix di_ is used to indicate classes that contain domain information, tbox_ to indicate toolbox classes for general use, and dt_ to indicate classes used in the decision tree. Note that the di_ classes can be used in other classification and clustering algorithms, not just decision trees. A brief description of the classes is as follows:

- di_FeatureValue: This contains either a nominal (discrete) feature or a numeric (continuous) feature, but never both at the same time.
- di_InstanceInfo: This contains the number of features, the name of the features and their type for a data instance.
- di_Instance: This contains the features for a data instance. It is typically used in conjunction with an object from the di_InstanceInfo class.
- di_InstanceArray: This can be used for the training set, where each data instance has several features or even for the feature lists that contain only a single feature and are created in the first part of the axis-parallel decision tree.
- tbox_NominalHistogram: This creates a histogram for nominal data.
- dt_SplitCriterion: This abstract base class represents the split criterion used at each node. The derived classes implement the specific criterion, enabling us to easily add new criteria without any modification to the rest of the code. The same split criterion is used in the entire decision tree.
- dt_SplitFinder: This base class represents the approach used to find the split - whether axis-parallel, oblique, CART-LC, etc. The derived classes implement the actual determination of the split. The SplitFinder used at any node of the tree can vary within the tree. For example, an oblique SplitFinder may select an axis parallel approach if it is a better choice, or if the examples at the node are too few to justify an oblique split. Regardless of the choice of SplitFinder, the user independently selects the split criterion used to evaluate the split. We can exploit parallelism within the SplitFinder class.
- dt_TreeNode: This class contains the information on a node of the tree. It includes pointers to the InstanceArrays stored using a single feature at a time, the left- and right- hand sides of the split made at the node, the type of SplitFinder, the count statistics for each feature, and pointers to the children nodes created by the split. Once the split is determined using the SplitFinder, the TreeNode object is responsible for actually splitting the instances among the children node. Parallelism can be exploited within this class.
- dt_DecisionTree: This is the main class that creates, tests, and applies the tree. It can also print out the tree, save it to a file, and read it back from a file. Starting with a root TreeNode that contains the entire training set, it creates the child nodes by choosing the appropriate SplitFinder, using the SplitCriterion set by the user. The single sort that may be required by some SplitFinders is done at the beginning of the training of the decision tree. Parallelism is exploited within this class.

3.3 Challenges in Parallel Implementation

Supporting several different options in a parallel decision tree software can be challenging as the approach taken to efficiently implement one option could directly conflict with the efficient implementation of another option. An interesting case of this arises in the SplitFinder class. The ScalParC approach, which generates axis-parallel trees, sorts each feature at the beginning of the creation of the tree. As a result, the features in a single data instance can be spread across more than one processor. However, for oblique classifiers, we need all the features of a data instance in order to evaluate a split. If these features are spread across processors, the communication required could be extensive with an irregular pattern. This would suggest that for oblique splits,

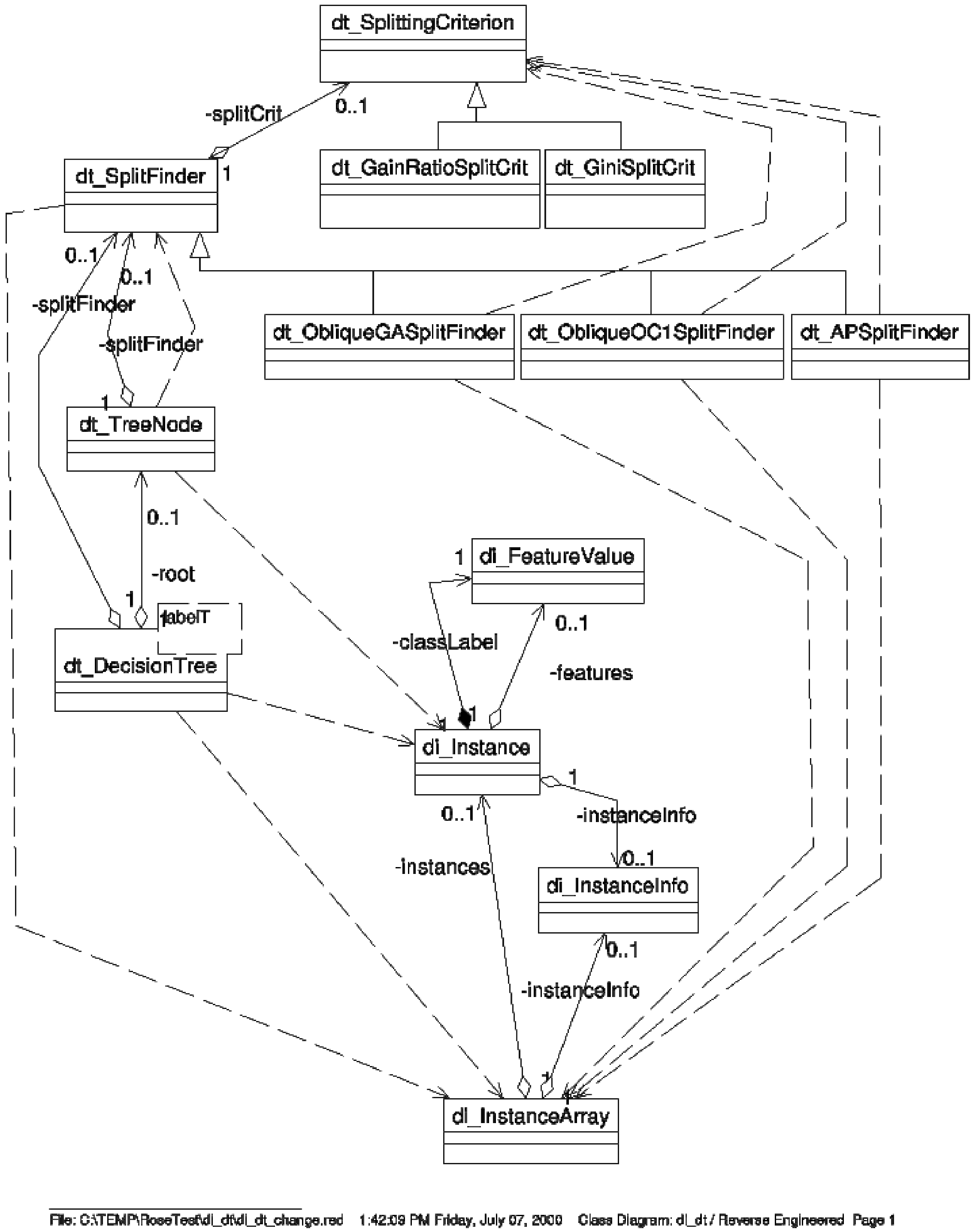
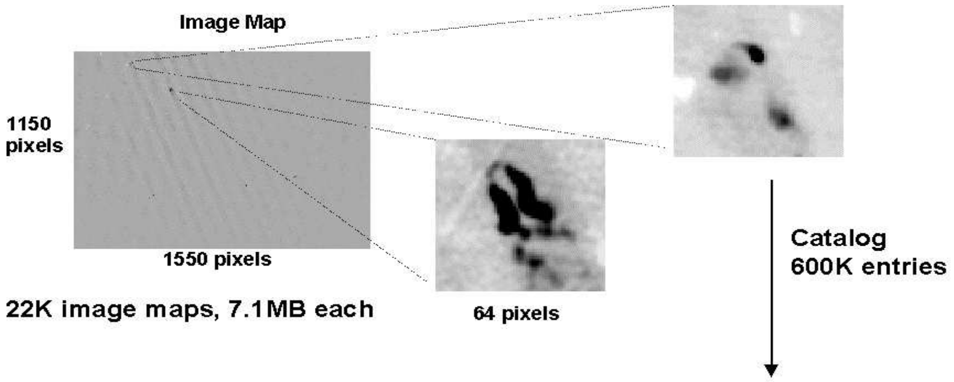


Figure 3: The UML Class Diagram for Decision Trees



RA	DEC	Peak Flux (mJy/bm)	Major Axis (arcsec)	Minor Axis (arcsec)	Position Angle (degrees)
00 56 25	-01 15 43	25.38	7.39	2.23	37.9
00 56 26	-01 15 57	5.50	18.30	14.29	94.2
00 56 24	-01 16 31	6.44	19.34	10.19	39.8

Figure 4: A FIRST image map with two bent-double galaxies and the catalog entries

we should not sort each of the features at the beginning. However, an oblique split requires the evaluation of an axis parallel as it selects the better of the two.

This gives rise to an interesting dilemma for oblique splits - should we sort each feature at the beginning or not? One option is to have two sets of features, one sorted and the other unsorted, even though it would almost double the memory requirements. The other option is to avoid the extensive communication that would result from sorting the data by approximating an axis-parallel split efficiently on un-sorted data, as described next.

The normal way to determine the best axis parallel split, is to first sort the values for a feature, and then evaluate a split by considering the split point as mid-way between two consecutive feature values. The best split across all features is chosen as the best split at a node. To approximate this, we can generate a histogram for each of the features, and select as a split value the boundary value of each bin in the histogram. If the histogram kept track of the count statistics for each class in a bin, we could use this information to select the best split based on any splitting criterion. By choosing the bin widths appropriately, we can obtain a good approximation to the axis-parallel split. Related work using this approach is described in [1].

Another issue to consider in the parallel implementation of decision trees is the inherent serial nature of some of the algorithms. For example, in the CART-LC and OC1 SplitFinders, the algorithm cycles through each of the coefficients of the hyperplane to find the best value for the coefficient, while keeping the other coefficients constant. This makes the algorithms inherently serial. The Oblique-EA algorithms do not have this drawback as they consider all coefficients of the hyperplane at a time. In addition, in the OC1 algorithm, the work done at each node of the tree may vary based on the initial (random) choices made by the algorithm at each node. This could make load balancing difficult. In contrast, the total work done at each node of the oblique-EA algorithms is fixed, making load balancing easier.

A different issue we need to address is the implementation of parallel decision trees on clusters of SMPs, where we could benefit from both distributed and shared memory programming. This could give rise to interesting solution approaches especially when evolutionary algorithms are used [4].

4. MINING THE FIRST DATASET

In this section, we describe an application in astronomy that we are using as a test-bed for our work. The Faint Images of the Radio Sky at Twenty-cm (FIRST) [2], is an astronomical survey using the Very Large Array at the National Radio Astronomy Observatory (http://sundog.stsci.edu). The FIRST astronomers are surveying more than 10,000 square degrees of the sky, to a flux density limit of 1.0 mJy (milli-Jansky). With the data collected through 1998, FIRST has covered about 6,000 square degrees, producing more than 20,000 two-million pixel images. At a threshold of 1 mJy, there are approximately 90 radio-emitting galaxies, or radio sources, in a typical square degree.

While radio sources exhibit a wide range of morphological types, the FIRST astronomers are particularly interested in galaxies with a bent-double morphology, as they indicate the presence of clusters of galaxies. Figure 4 has two images that were identified manually by scientists as bent-double galaxies. This visual inspection of the radio images, besides being very subjective, is also becoming increasingly infeasible as the survey grows in size. To address this problem, we are applying data mining techniques to the FIRST data to identify bent-double galaxies in a semi-automated way.

The data from the FIRST survey is available as image maps and a catalog. In Figure 4, we show an image map and

Algorithm	Parameter	Twoing	Gini	Info Gain
OC1-AP	Error	8.62 (0.58)	8.14 (0.54)	9.17 (0.67)
	Leaves	3.63 (0.41)	3.68 (0.42)	4.3 (0.37)
	Time	2.97 (0.07)	2.84 (0.06)	4.72 (0.09)
OC1	Error	12.76 (0.61)	13.16 (0.39)	14.29 (0.59)
	Leaves	2.9 (0.13)	2.86 (0.12)	2.96 (0.11)
	Time	44.73 (1.0)	252 (14.5)	116 (2.1)
OC1-CART	Error	13.13 (0.56)	12.51 (0.68)	13.12 (0.64)
	Leaves	2.61 (0.16)	2.82 (0.14)	2.69 (0.13)
	Time	10.37 (0.27)	9.22 (0.22)	26.64 (1.17)
Oblique-ES	Error	8.69 (0.58)	8.03 (0.59)	8.83 (0.59)
	Leaves	3.59 (0.41)	3.9 (0.40)	4.24 (0.36)
	Time	34.96 (0.92)	34.5 (0.86)	36.21 (0.63)
Oblique-GA	Error	10.76 (0.65)	8.65 (0.64)	11.96 (0.71)
	Leaves	3.54 (0.14)	3.54 (0.15)	3.71 (0.18)
	Time	162 (2.9)	112 (3.2)	170 (3.4)

Table 1: Comparison of different algorithms and split criteria on the FIRST data set.

the three catalog entries corresponding to one of the bent-doubles present in the image map. These large image maps are mostly "empty", that is, composed of background noise that appear as streaks in the image. The FIRST catalog [19] is obtained by processing an image map to fit two-dimensional elliptic Gaussians to each radio source. Each entry in the catalog corresponds to the information on a single Gaussian. This includes, among other things, the coordinates for the center of the Gaussian, the major and minor axes, the peak flux, and the position angle of the major axis (degrees counter-clock-wise from North).

To apply data mining to the FIRST data, we first extracted relevant features for the bent-doubles. From conversations with the FIRST astronomers, we learned that the catalog could be considered a good approximation to all but the most complex of radio sources. As a result, we could initially focus only on the catalog, without having to use the images as well. This was fortuitous as feature extraction from the images would have meant extensive processing in order to de-noise the images without affecting the signal, identify the galaxies, and then extract relevant features.

Our approach to feature extraction for bent-doubles is described in detail in an earlier work [6]. We first grouped the catalog entries that were close to each other, and then focused on those groups that consisted of two or three catalog entries. This was based on the observation that a single entry galaxy was unlikely to be a bent-double, while four or more entries in a galaxy would make it complex enough to be of interest to astronomers anyway. We next extracted a separate set of features for the two- and three-entry galaxies, focusing on features such as relative distances and angles between entries, that were likely to be robust and invariant to rotation, scaling, and translation. Separating the two- and three-entry galaxies enabled us to have uniform length feature vectors for each. However, it also meant that a small training set (313 examples) was split further into smaller training sets of 118 examples for two-entry and 195 examples for three-entry sources, respectively.

Our initial experimentation with axis-parallel decision trees on the bent-double problem indicated that for the three-entry case the error was approximately 9%, but the error for the two-entry case was closer to 20%. We suspect that the poor performance of the two-entry case could be the result of a small training set, information in the catalog that is not representative of the images, and possible inclusion of non-relevant features while missing the relevant ones. We are investigating this issue further. For this paper, we will focus on the three-entry radio sources.

5. EXPERIMENTAL RESULTS

In this section, we describe some preliminary results for the bent-double problem using our decision tree software. As mentioned earlier, our training set has 195 examples. Each example is described by 103 numeric features. In the FIRST domain, there are two possible classes: an instance describes either a bent or a non-bent galaxy. We evaluated the performance of the five different SplitFinder algorithms described in Section 3.1 using ten, ten-fold cross-validation experiments. For each algorithm, we tried three different impurity measures: twoing, Gini index, and the information gain. The results are summarized in Table 1. The table shows the mean error rate, the average number of leaves of the trees found (after pruning), and the average time (in seconds). The numbers in parenthesis are the standard errors for each result.

The results show that the oblique-EA algorithms have better accuracy than the other oblique algorithms. Oblique-ES and the Oblique-GA consistently give better solutions than OC1 and OC1-CART (the version of CART-LC implemented in the OC1 software). The Oblique-ES trees have approximately 30–40% fewer classification errors than the existing oblique DTs, and the Oblique-GA has approximately 15–35% fewer errors. In all cases, the evolutionary algorithms are faster than the original OC1, but CART is the fastest overall oblique DT. However, the accuracy of the best oblique trees is not significantly different than the axis-parallel trees, and it seems that AP trees are a good choice in the FIRST domain.

```
rs3_core_angle > 170.4:
:...cec_ellipticity <= 2.116: 1 (13.0)
:   cec_ellipticity > 2.116: 5 (2.0)
rs3_core_angle <= 170.4:
:...pairac_rel_dist <= 9.423: 5 (143.0)
    pairac_rel_dist > 9.423:
    :...pairab_angle_geom <= 58.6: 5 (4.0/1.0)
        pairab_angle_geom > 58.6:
        :...cec_rms <= 0.137: 5 (5.0/2.0)
            cec_rms > 0.137: 1 (9.0)
```

Figure 5: An axis-parallel decision tree for the FIRST domain.

The table also shows that the solution accuracy improves slightly when the Gini index is used to evaluate the splits, except for OC1 where the Twoing rule gives the best results. The differences in accuracy are not significant, however, except in the case of the GA. The tree sizes also seem unaffected by the split evaluation criteria, but in some cases there are large differences in the training time. The Gini index seems to need the shortest training times for all algorithms except OC1, but some of the differences are not significant.

The differences in training time may be caused by several factors, which we are currently investigating. We suspect that some combinations of algorithms and split impurity measures result in not-so-good splits near the root of the tree for this particular problem. If this were the case, the algorithm would continue to split the data until it finds relatively pure partitions, which requires additional hyperplane evaluations and longer training times. If good splits are found near the root, the algorithm would stop earlier, requiring a shorter training time. Note that the slowest algorithms are also the least accurate, which is consistent with our hypothesis.

We are also trying to better understand why the axis-parallel trees work so well in this problem. Figure 5 shows a typical axis-parallel tree. The output lists the feature selected at each node and the value it is compared against. The number after the colon indicates a leaf node, where the number is the class assigned to the leaf (5 is a bent double, while 1 is a non-bent double). The numbers in parenthesis at each leaf node, (a/b), indicate that of the total samples a at that node, $(a-b)$ samples belonged to the class assigned to the leaf node, while b samples were of the class not assigned to the leaf node. This tree has a depth of four, with six leaves.

Our conversations with the FIRST astronomers indicated that the axis-parallel tree agreed with their approach to the identification of a bent-double. Further, the features selected by the tree, such as the relative distances and angles, were very relevant. Note also that the leaf nodes tend to be very pure, that is, they are composed of only one class. Also, a substantial percentage of the examples (90% = 158 out of 176) have been correctly classified by a tree of depth two. In addition, 81% of the examples (143 out of 176), can be classified as bent-doubles based solely on two features using axis parallel hyperplanes. All these observations seem to indicate that the classification of bent-doubles, based on the features we have extracted, is a problem very well suited for axis-parallel trees.

These observations may also give us a possible approach to selecting a decision tree algorithm for a problem. Since axis-parallel trees take relatively little time to generate, even for large trees, we can easily generate one to understand how it partitions the feature space. If it creates a very large tree, with few samples at each leaf, and leaves with low purity, an oblique classifier may be a better solution. In such cases, oblique classifiers based on evolutionary algorithms, such as the ones proposed in this paper, can be an attractive alternative to the standard oblique classifier.

6. SUMMARY

In this paper, we discussed our design goals for a parallel object-oriented software toolkit for mining scientific data sets. We have described how our design can meet the diverse needs of our applications. Focusing on a specific example, namely decision trees, we presented some of the challenges we face as we implement several different variants of decision trees within a parallel framework. We also described briefly our approach to addressing these challenges and our experiences with an astronomy data set.

7. ACKNOWLEDGEMENTS

We acknowledge the contributions of the rest of the Sapphire project team: Chuck Baldwin, Imola Fodor, and Nu Ai Tang.

UCRL-JC-138973: This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

8. REFERENCES

[1] Alsabti, K., Ranka, S., and Singh, V. CLOUDS: A decision tree classifier for large datasets. In *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining* (1998).

[2] Becker, R., White, R., and Helfand, D. J. The FIRST Survey: Faint Images of the Radio Sky at Twenty-cm. *Astrophysical Journal 450* (1995), 559.

[3] Breiman, L., Friedman, J., Olshen, R. A., and Stone, C. *Classification and Regression Trees.* CRC Press, Boca Raton, Florida, 1984.

[4] Cantú-Paz, E. A survey of parallel genetic algorithms. *Calculateurs Paralleles, Reseaux et Systems Repartis 10*, 2 (1998), 141–171.

[5] Cantú-Paz, E., and Kamath, C. Using evolutionary algorithms to induce oblique decision trees. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), July 2000* (2000).

[6] Fodor, I., Cantú-Paz, E., Kamath, C., and Tang, N. Finding Bent-Double Radio Galaxies: A Case Study in Data Mining. In *Proceedings of the Interface: Computer Science and Statistics Sympos ium* (2000), vol. 33.

[7] Joshi, M. V., Karypis, G., and Kumar, V. ScalParC: A new scalable and efficient parallel classification algorithm for mining large datasets. In *Proceedings of the 12th International Parallel Processing Symposium* (1998).

[8] Kamath, C., and R., M. Scalable data mining through fine-grained parallelism: the present and the future. In *Advances in Distributed and Parallel Knowledge Discovery*, H. Kargupta and P. Chan, Eds. AAAI/MIT Press, 2000.

[9] MPI Forum. *http://www.mpi-forum.org*.

[10] Murthy, K. V. S. *On Growing Better Decision Trees from Data*. PhD thesis, Johns Hopkins University, 1997.

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

[12] Quinlan, J. R. Induction of decision trees. *Machine Learning 1* (1986), 81–106.

[13] Quinlan, J. R., and Rivest, R. Inferring decision trees using the minimum description length principle. *Information and Computation 80*, 3 (1989), 227–248.

[14] Sapphire: Large-Scale Data Mining and Pattern Recognition. *http://www.llnl.gov/casc/sapphire*.

[15] Shafer, J., Agrawal, R., and Mehta, M. SPRINT: A scalable parallel classifier for data mining. In *Proceedings of the 22nd Conference on Very Large Data Bases* (1996).

[16] Sreenivas, M., K., A., and Ranka, S. Parallel out-of-core decision tree classifiers. In *Advances in Distributed and Parallel Knowledge Discovery*, H. Kargupta and P. Chan, Eds. AAAI/MIT Press, 2000.

[17] Srivastava, A., Han, E. H., Kumar, V., and Singh, V. Parallel formulations of decision-tree classification algorithms. *To appear in Data Mining and Knowledge Discovery; An International Journal* (1999).

[18] Object Management Group: The Unified Modeling Language. *http://www.omg.org*.

[19] White, R., Becker, R., Helfand, D., and Gregg, M. A Catalog of 1.4GHz Radio Sources from the FIRST Survey. *Astrophysical Journal 475* (1997), 479.

[20] Zaki, M. J., Ho, C. T., and Agrawal, R. Parallel Classification on SMP Systems. In *Proceedings of the 1st Workshop on High Performance Data Mining* (1998).