4th International Workshop on Parallel and Distributed Data Mining April 27th, 2001 San Francisco, CA, USA

in conjunction with
15th International Parallel and Distributed Processing Symposium

Mohammed J. Zaki, Vipin Kumar, and David Skillicorn

Preface

As the volume of data increases, it is clear that both parallel and distributed data mining techniques are required to make the whole knowledge discovery process scalable and interactive. This workshop called for papers on high performance parallel and distributed methods, as well as mining on distributed and heterogeneous datasets. Topics of interest included:

- Efficient, scalable, disk-based, parallel and distributed algorithms for large-scale data mining tasks
- New algorithms for common data mining methods such as association rules, sequences, classification, clustering, deviation detection, etc.
- Pre-processing and post-processing operations like sampling, feature selection, data reduction and transformation, rule grouping and pruning, etc.
- Incremental, exploratory and interactive mining
- Meta-mining, coping with distributed and/or heterogeneous datasets
- Integration of mining with parallel/distributed databases and data-warehouses
- Mining non-traditional datasets, such as large scientific databases
- Frameworks for KDD systems, and parallel or distributed mining
- Agent based approaches for PDDM
- Applications of PDDM in business, science, engineering, medicine, and other disciplines
- Theoretical foundation of PDDM

This is the 4th workshop on this theme held annually in conjunction with the IPDPS conference. The first three workshops went under the name "High Performance Data Mining," and were held at Orlando (HPDM'98), San Juan (HPDM'99) and Cancun (HPDM'00). In keeping with the growing

Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS '01)
1530-2075/01 $10.00 © 2001 **IEEE**

popularity and international scope of this field, this workshop was renamed as the "International Workshop on Parallel and Distributed Data Mining".

These proceedings contain 5 papers that were accepted for presentation at the workshop. Each paper was reviewed by two or three members of the program committee. In keeping with the spirit of the workshop some of these papers also represent work-in-progress. In all cases, however, the workshop program highlights avenues of active research in high performance data mining.

We would like to thank all the authors and attendees for contributing to the success of the workshop. Special thanks are due to the program committee for help in reviewing the submissions.

Mohammed J. Zaki
Vipin Kumar
David B. Skillicorn
Editors

Workshop Co-Chairs

Mohammed J. Zaki (Rensselaer Polytechnic Institute, USA)
Vipin Kumar (University of Minnesota, USA)
David B. Skillicorn (Queens University, Canada)

Program Committee

Philip K. Chan (Florida Institute of Technology, USA)
David Cheung (University of Hong Kong, Hong Kong)
Alok Choudhary (Northwestern University, USA)
Alex A. Freitas (Pontifical Catholic University of Parana, Brazil)
Johannes Gherke (Cornell University, USA)
Robert Grossman (University of Illinois-Chicago, USA)
Yike Guo (Imperial College, UK)
Howard Ho (IBM Almaden Research Center, USA)
Chandrika Kamath (Lawrence Livermore National Labs., USA)
Hillol Kargupta (University of Maryland-Baltimore County, USA)
Masaru Kitsuregawa (University of Tokyo, Japan)
William Maniatty (University at Albany-SUNY, USA)
Charles Musick, (iKuni Inc, USA)
Yi Pan (Georgia State University, USA)
Srinivasan Parthasarathy (Ohio State University, USA)
Foster Provost (New York University, USA)
Arno Siebes (CWI: Centrum voor Wiskunde en Informatica, Netherlands)
Domenico Talia (ISI-CNR: Institute of Systems Analysis and Information Technology, Italy)
Albert Zomaya (University of Western Australia, Australia)

Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS '01)
1530-2075/01 $10.00 © 2001 **IEEE**

Towards a Parallel Data Mining Toolbox

Peter Christen* Markus Hegland Ole M. Nielsen Stephen Roberts Peter Strazdins
Tatiana Semenova
Australian National University, Canberra, ACT 0200, Australia

Irfan Altas
Charles Sturt University, Wagga Wagga, NSW 2678, Australia

Timothy Hancock
James Cook University, Townsville, QLD 4811, Australia

Abstract

This paper presents research projects tackling two aspects in data mining. First, a toolbox is discussed that allows flexible and interactive data exploration, analysis and presentation using the scripting language Python. The advantages of this toolbox are that it provides the functionality to process multiple SQL queries in parallel, and enables fast data retrieval using a supervised caching mechanism for commonly used queries. These two facets of the toolbox allow for fast, efficient data access reducing the time spent on data exploration, preparation and analysis.

Secondly, an approach to predictive modelling is presented that leads to scalable parallel algorithms for high dimensional data collections. This is an essential requirement for data mining algorithms as those that do not scale linearly with the data size are infeasible. These algorithms are implemented in parallel and achieve an almost ideal speedup for their respective implementations.

One aim of the presented research is to integrate and combine these two different aspects of data mining into an efficient but flexible data mining toolbox that allows the experienced data miner to attack large scale problems interactively or with batch processing.

1 Introduction

There is much ongoing research in sophisticated algorithms for data mining purposes. Examples include predictive modelling, genetic algorithms, neural networks, decision trees, association rules, and many more. However, it is generally accepted that it is not possible to apply such algorithms without a careful data preparation and data understanding process, which may often dominate the actual data mining activities [8, 20]. It is also rarely feasible to use off-the-shelf data mining software and expect useful results without a substantial amount of data insight. In addition, data miners working as consultants are often presented with datasets from an unfamiliar domain and need to get a good feel for the data and the domain prior to any "real" data mining. The ease of initial data exploration and preprocessing may well hold the key to successful data mining results later in a project. These processes are highly interactive: The data miner investigates the data and extracts subsets of attributes or transactions to be mined and conducts experiments that lead to new ideas and questions requiring further exploration. Fast and flexible data querying and aggregation are therefore mandatory.

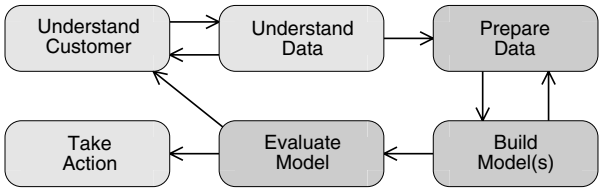

Figure 1. The data mining process

Data mining is an iterative process (Figure 1), as various steps may have to be repeated several times until useful and valuable knowledge is found, e.g. the same mining algorithm is used on different subsets of a data collection to compare different outcomes. A lot of effort in a data mining project is often spent with time consuming routine tasks. A caching of intermediate results can thus shorten response times tremendously and help the data miner to concentrate on knowledge extraction.

Today data collections have the size of Gigabytes and

*Corresponding author, E-Mail: Peter.Christen@anu.edu.au

Terabytes, and the first Petabyte databases are appearing in science [10]. Data mining tools thus have to be able to handle large amounts of data efficiently and they also need to be scalable with the increasing size of data collections. Therefore, algorithms which do not scale linearly with the data size are not feasible. Additionally, the dimensionality of datasets is increasing, which is a major challenge for many algorithms as their complexity grows exponentially with the dimension of a dataset. This has been called the *curse of dimensionality* [13]. Parallel processing can help both to tackle larger problems and to get reasonable response times. It is not only that more computing power becomes available, but equally important is the increased I/O bandwidth and larger main memory provided by most parallel machines.

A further challenge in real world data mining projects is the various data formats that have to be dealt with, like relational databases, flat text files, non-portable binary files or data downloaded from the Web. A flexible *middleware* layer can unify the view of different data collections and facilitate the application of various tools and algorithms.

This paper presents ideas and methods that tackle many of the described challenges. A toolbox approach using the scripting language Python [5] allows flexible data exploration, while parallel predictive modelling algorithms that are scalable both with the number of attributes of a data collection as well as the number of used processing nodes, allow mining of high-dimensional data sets. The toolbox *DMtools* [18] is currently under development and a predecessor has successfully been applied in real-world data mining projects under the ACSys CRC grant[1]. The toolbox assists our research group in all stages of data mining projects, starting from data preprocessing, analysis and simple summaries up to visualisation and report generation. In Section 2 related work in the relevant areas is presented, and Section 3 presents our toolbox approach in more detail. Section 4 discusses our scalable parallel algorithms for predictive modelling, and Section 5 gives an outlook to future work.

2 Related work

There are several projects describing toolbox like approaches to data exploration. The authors of the IDEA (Interactive Data Exploration and Analysis) system [21] identify five general user requirements for data exploration: Querying (the selection of a subset of data according to the values of one or more attributes), segmenting (splitting the data into non-overlapping sub-sets), summary information (like counts or averages), integration of external tools and applications, and history mechanisms. The IDEA framework allows quick data analysis on a sampled sub-set of the data with the possibility to re-run the same analysis later on the complete dataset. IDEA runs on a PC, with the user interacting on a graphical interface.

Another approach used in the Control [14] project is to trade quality and accuracy for interactive response times, in a way that the system quickly returns a rough approximation of a result that is refined continuously. The user can therefore get a glimpse at the final result very quickly and use this information to change the ongoing process. The Control system, among others, includes tools for interactive data aggregation, visualisation and data mining.

Database research and data mining are two related fields and there are many publications dealing with both areas. An overview of database mining is given in [7]. According to the authors the efficient handling of data stored in relational databases is crucial because most available data is in a relational form. Scalable and efficient algorithms are one of the challenges, as is the development of high-level query languages and user interfaces so data mining tasks can be specified by non-experts. One of the identified key requirements is interactivity. The possibilities to interactively analyse data collections, to refine data mining requests, to deepen the analysis and to change the focus should be encouraged, because it is often difficult to predict what exactly could be discovered from a dataset. Interactive data mining is also needed if transformation and manipulation of data are necessary, or if different subsets of a data collection are to be examined. To be able to deal with the huge amounts of data available, parallel and distributed data mining algorithms are important, as is the possibility to mine information from different sources of data.

Parallel data mining is a hot research topic (see [24] for recent research papers), as the need for parallel processing is clearly given by the huge and increasing data collections available. The requirements for parallel KDD systems [17] include not only parallel scalable hardware platforms, parallel I/O and databases, and parallel data mining algorithms, but also frameworks for rapid algorithm development and evaluation. Issues like security, fault tolerance, heterogeneous data access and representation, quality of service, pricing and portability have to be addressed as well. Large-scale parallel KDD systems should support the entire KDD process, including pre- and post-processing.

3 A toolbox for data mining

Using a portable, flexible, and easy to use toolbox can not only facilitate the data exploration phase of a data mining project, it can also help unifying the data access with a middleware library to integrate the access of different data sources to the data mining applications. Thus it forms the

[1] ACSys CRC stands for 'Advanced Computational Systems Collaborative Research Centre' and the data mining consultancies were conducted at the Australian National University (ANU) in collaboration with the Commonwealth Scientific and Industrial Research Organisation (CSIRO)

framework for the application of a suite of more sophisticated data mining algorithms. The command line driven approach of a toolbox – which is also used in packages like Matlab or Mathematica – is maybe not suited for a novice user, but for the experienced data miner it provides a more powerful and flexible tool than a graphical user interface.

The *DMtools* [18] are based on the scripting language *Python* [5], chosen since it has proven to be an excellent tool for rapid code development. It allows for efficient handling of large datasets, it is flexible and easily extensible. Functions and routines can be used as templates which can be changed and extended as needed by the user to do more customised analysis tasks. Having a new data exploration idea in mind, the data miner can implement a rapid prototype very easily by writing a script using the functions provided by our toolbox.

Because many data collections are stored in relational databases, it is important that such data can be accessed efficiently by data mining applications [6]. Furthermore, databases using SQL are a standard tool for storing and accessing transactional data in a safe and well-defined manner. However, both complex queries and transmissions of large data quantities tend to be prohibitively slow. For our toolbox we follow another route: Only simple queries (e.g. no joins) are sent to the database server, and the results are cached and processed within the toolbox. The Python database API [15] allows us to access a relational database by SQL queries. Currently, we are using MySQL [23] for the underlying database engine, but Python modules for other database servers are available as well. Both MySQL and Python are licensed as free software and enjoy very good support from a large user community. In addition, both products are very efficient and robust.

3.1 Toolbox architecture

In our toolbox the ease of SQL queries and the safety of relational databases are combined with the efficiency of binary file access and the flexibility of object-oriented programming languages in an architecture as shown in Figure 2. Based on relational databases, flat files, the Web, or any other data source a *Data Manager* deals with retrieval, caching and storage of data. It provides routines to execute an arbitrary SQL query and to read and write binary and text files. The two important core components of this layer are its transparent caching mechanism and its parallel database interface which intercepts SQL queries and parallelises them on-the-fly. The *Aggregation* module implements a library of Python routines taking care of simple data exploration, statistical computations, and aggregation of raw data. The *Modelling* module contains functions for parallel predictive modelling, clustering, and generation of association rules. The *Report* module provides visualisation

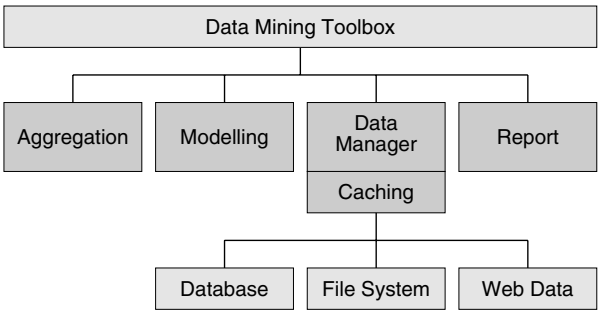

Figure 2. Architecture of DMtools

and allows facilities for simple automatic report generation. Complex domain-specific functions and end-user applications can then be written in terms of the scripting language making use of the available modules.

3.2 Caching and database parallelism

Caching of function results is a core technology used throughout our data mining toolbox. We have developed a methodology for *supervised* caching of function results as opposed to the more common (and also very useful) *automatic* disk caching provided by most operating systems and Web browsers.

Like automatic disk caching, supervised caching trades space for time, but the approach we use is one where time consuming operations such as database queries or complex functions are intercepted, evaluated and the resulting objects are made persistent for rapid retrieval at a later stage. We have observed that most of these time consuming functions tend to be called repetitively with the same arguments. Thus, instead of computing them every time, the cached results are returned when available, leading to substantial time savings. The repetitiveness is even more pronounced when the toolbox cache is shared among many users, a feature we use extensively.

This type of caching is particularly useful for computationally intensive functions with few frequently used combinations of input arguments. Note that if the inputs or outputs are very large, caching might not save time because disk access may dominate the execution time. Supervised caching is invoked in the toolbox by explicitly applying it to chosen functions. For a given Python function of the form `T = func(arg1,...,argn)` caching in its simplest form is invoked by replacing the function call with the following call: `T = cache(func,(arg1,...,argn))`. This structure has been applied in the *Data Manager* module of the toolbox so using the top level toolbox routines will utilise caching transparently. For example, most of the SQL queries that are automatically generated by the toolbox are cached in this fashion. Generating queries automatically in-

creases the chance of cache hits as opposed to queries written by the end user because of their inherent uniformity. In addition to this, caching can be used in code development for quick retrieval of precomputed results. For example, if a result is obtained by automatically crawling the Web and parsing HTML or XML pages, caching will help in retrieving the same information later – even if a Web server is unserviceable.

An example query used in a health data mining project that extracts all patients belonging to a particular group together with a count of their transactions required on the average 489 seconds worth of CPU time on a Sun Enterprise server and the result took up about 200 Kilobytes of memory. After having cached this query, subsequent loading takes 0.22 seconds – more than 2,000 times faster than the computing time. This particular function was hit 168 times in a fortnight saving four users a total of 23 hours of waiting.

If a function definition changes after a result has been cached, or if the result depends on other files, wrong results may occur when using caching in its simplest form. The caching utility therefore supports specification of explicit dependencies in the form of a file list, which, if modified, triggers a recomputation of the cached function result.

If a database server allows parallel execution of independent queries, we use this parallelism within our toolbox by sending a list of queries to the database server and process the returned results sequentially. This is very efficient if the queries take a long time to proceed, but only return a small result list.

Example 3.1 *Caching of XML documents*

Supervised caching is used extensively for database querying but is by no means restricted to this. Caching has proven to be useful in other aspects of the data mining toolbox. An example is a Web application built on top of the toolbox which allows managers to explore and rank branches according to one or more user-defined features such as *Annual revenue*, *Number of customers serviced relative to total population*, or *Average sales per customer*. The underlying data is historical sales transaction data which is updated monthly, so features need only be computed once for new data when it is added to the database. Because the data is static, cached features are never recomputed and the application can therefore make heavy use of the cached database queries. Moreover, no matter how complicated a feature is, it can be retrieved as quickly as any other feature once it has been cached. In addition, the Web application is configured through an XML document defining the data model and describing how to compute the features. The XML document must be read by the toolbox, parsed and converted into appropriate Python structures prior to any computations. Because response time is paramount in an interactive application, parsing and interpretation of XML is prohibitive, but by using the caching module, the resulting Python structures are cached and retrieved quickly enough for the interactive application. The caching function was made dependent on the XML file itself, so that all structures are recomputed whenever the XML file has been edited – for example to modify an existing feature-definition, add a new month, or change the data description. Below is a code snippet from the Web application. The XML configuration file is assumed to reside in `sales.xml`. The parser which builds Python structures from XML is called `parse_config` and it takes the XML filename as input. To cache this function, instead of the call `(feature_list, branch_list) = parse_config(filename)` we write:

```
filename = "sales.xml"
(feature_list, branch_list) =     \
  cache(parse_config, filename,   \
        dependencies = filename)
```

3.3 Integration of parallel applications

One aim of our research is to integrate parallel applications into the toolbox to enable fast and efficient execution of large and complex data mining tasks, so the data miner is able to attack large problems interactively or with batch processing. The toolbox also gives a common interface to various data mining algorithms, whereby the details of the parallel application and architecture (like starting and working with a parallel environment) are hidden from the user.

Using a scripting language like Python to control parallel applications has already been used for steering [4, 19] of scientific applications, where a user can change parameters at run-time to control the behaviour of long-running simulations like molecular dynamics applications.

In our toolbox, we use the standard Python interpreter to start parallel applications with a dynamically generated system call. The Unix command `system` is executed through Python to invoke an MPI [12] program like in the following example:

```
mpi_str = "mpirun -np " + str(num_proc) + \
          predmodel " + arg_str + result_str
os.system(mpi_str)
```

The string `arg_str` contains the input arguments and `result_str` contains the name of the result file. A Python wrapper code converts toolbox objects (like lists or dictionaries) into a format which is processable by the parallel application, and creates configuration and data files as necessary. The parallel application gets its input parameters from the Python script at the command line, and then loads and processes the requested data files. Results are saved to files by the parallel program and read by the toolbox for further processing. Time consuming data mining algorithms

can therefore be run in parallel, while the Python toolbox handles the more high level aspects of data mining like preprocessing and visualisation.

4 Scalable parallel predictive modelling

Algorithms applied in data mining have to deal with two major challenges: Large datasets and high dimensionality. It has also been suggested that the size of databases in an average company doubles every 18 months [3] which is similar to the growth of hardware performance according to Moore's law. Consequently, data mining algorithms have to be able to scale from smaller to larger data sizes when more data becomes available. The complexity of data is also growing as more attributes tend to be logged in each record. Data mining algorithms must, therefore, be able to handle high dimensions in order to process such datasets, and algorithms which do not scale linearly with data size and dimension are not feasible.

An important technique applied in data mining is predictive modelling. A predictive model in some way describes the average behaviour of a data set, and can be used to find data records that lie outside of the expected behaviour. These *outliers* often have simple natural explanations but, in some cases, may be linked to fraudulent behaviour.

A predictive model is described by a function $y = f(x_1, \ldots, x_d)$ from the set, T, of attribute vectors of dimension d in the response set, S. If S is a finite set (often $S = \{0, 1\}$), the determination of f is a *classification problem* and if S is the set of real numbers, one speaks of *regression*. In the following it will mainly be assumed that all the attributes x_i as well as y are real values and we set $\mathbf{x} = (x_1, \ldots, x_d)^T$.

In many applications, the response variable y is known to depend in a smooth way on the values of the attributes, so it is natural to compute f as a least squares approximation to the data with an additional smoothness component imposed. In this paper, we state the problem formally as follows. Given n data records $(\mathbf{x}^{(i)}, y^{(i)}), i = 1, \ldots, n$ where $\mathbf{x}^{(i)} \in \Omega$ with $\Omega = [0, 1]^d$ (the d-dimensional unit cube), we wish to minimise the following functional subject to some constraints:

$$J_\alpha(f) = \sum_{i=1}^{n} (f(\mathbf{x}^{(i)}) - y^{(i)})^2 + \alpha \int_\Omega |\mathcal{L} f(\mathbf{x})|^2 \, d\mathbf{x} \quad (1)$$

where α is the smoothing parameter and $\mathcal{L}$ is a differential operator whose different choices may lead to different approximation techniques. The smoothing parameter α controls the trade-off between smoothness and fit. One can choose different function spaces to approximate the minimiser f of equation (1).

We have developed three different methods [9] to approximate the minimiser of Equation (1). TPSFEM uses piecewise multilinear finite elements and gives the most accurate approximation at the highest computational costs; HISURF is based on interpolatory wavelets which provides good approximations at reasonable costs; and ADDFIT implements additive models which have the lowest costs but give the coarsest approximation. The three methods differ in how well they approximate f and more importantly in their algorithmic complexities, but all three consist of the following two steps:

1. **Assembly**: An $m \times m$ symmetric matrix $\mathbf{A}$ and a corresponding $m \times 1$ vector $\mathbf{b}$ are assembled whose structures depend on the chosen method, but whose dimension m is independent on the number of data records n. For the TPSFEM method, the matrix structure is sparse with 3^d filled diagonals (d the dimensionality of the dataset), and for both HISURF and ADDFIT we store dense matrices. The size m of the assembled linear system depends on the total number of categories for categorical variables and on the resolution of the finite element grid for continuous variables. The assembly step requires access to all n data records once only, and it can be organised such that the amount of computational work is linear in n. As usually $m << n$ this step can be interpreted as a reduction operation on the original data. Note that the assembly of the matrices coming from the smoothing part of Equation (1) and constraints do not require accessing the data at all. These matrices have similar sizes to $\mathbf{A}$.

2. **Solving**: This step assembles the $m \times m$ matrices coming from the smoothing part of Equation (1) and solves the entire linear system. It does not involve the n data records and the computational work depends only on m, typically as $O(m^3)$ for the dense and $O(m)$ for the sparse equations.

Note that for large n step 1 will dominate whereas for large m step 2 will dominate. As the number of data records n is usually very large for data mining applications, the overall complexity is mainly determined by n.

The process of assembling the linear systems has the same structure for all three methods. For each data record, some nonzero elements are added into the matrix and vector. The number of nonzero elements per data record is $O(d)$, forming the normal equations matrix $\mathbf{A}$ is thus of order $O(d^2)$ for each data record. The total complexity of assembling n data records sequentially is therefore:

$$T_{assem}(1) = O(d^2 n) \quad (2)$$

The assembly of data records into the linear system is additive and thus each data record can be assembled independently from all others. By reading a fraction n/p of the

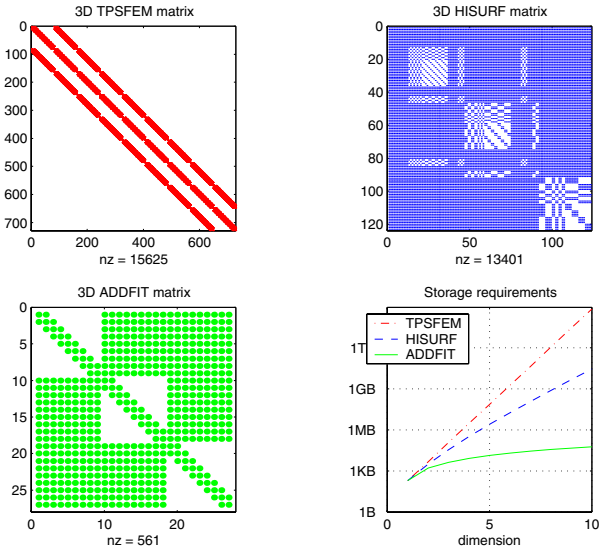

Figure 3. Matrix structure (3D) and complexity

dataset in parallel each processing node can assemble a local linear system without communication, so the parallel complexity of the assembly process on p nodes becomes:

$$T_{assem}(p) = O\left(\frac{d^2 n}{p}\right) \tag{3}$$

The complete matrix data structure has to be stored on each node, as every data record can contribute nonzero elements anywhere in the matrix. The linear system is therefore distributed, but not replicated, on the nodes and the final linear system is the sum of all local linear systems.

Figure 3 shows the structure of the assembled data matrix for the three methods for a 3D-problem with a grid resolution of nine points in each dimension. The lower right graph gives the amount of memory needed for a 1D to 10D problem (again with a grid resolution of nine points in each dimension). One can clearly see the limitation of the TPSFEM and HISURF methods due to their storage requirements.

The assembly process without communication is limited by the available amount of main memory. For matrices that are too large, a more complex assembly has to be applied (not yet implemented), where the matrix data has to be distributed in a memory-scalable way. A blocking structure of reading and redistributing data will be used.

4.1 Parallel implementation

At the time of writing, the assembly phase has been implemented for ADDFIT in ANSI C and using MPI [12] for communication. As the basic assembly structure is the same

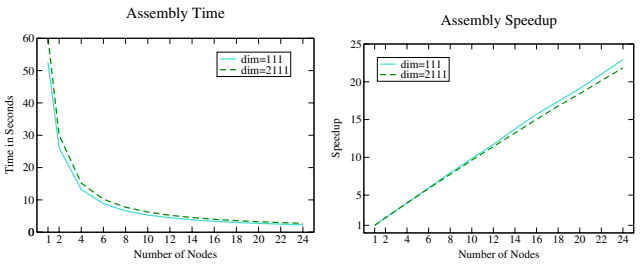

Figure 4. Assembly on Beowulf Linux cluster

for all three methods, it is simple to include analogous routines for TPSFEM and HISURF: The matrix data structures and the assembly of data records are the only parts that have to be changed.

First timing tests with synthetic datasets (consisting of 5 million records with 10 attributes each) show an almost ideal speedup for different matrix dimensions. As an example Figure 4 illustrates the times and speedup achieved on the 196 processor Beowulf Linux cluster *Bunyip* at the Australian National University [1]. This cluster is built with 98 dual 550 MHz Pentium III nodes, each equipped with 384 Megabytes of RAM (total about 36 Gigabytes), 13 Gigabytes of disk space (total 1.3 Terabytes) and 3×100 MBit/s fast Ethernet cards. Logically 96 nodes are connected in four groups of 24 nodes arranged as a tetrahedron with a group of nodes at each vertex. Two nodes are designated as servers. For our tests we only used one group (i.e. up to 24 nodes), whereas all data files have been distributed onto local disks. The results in Figure 4 show the times used for the assembly and redistribution of two linear systems of different size (corresponding to a different number of used attributes). The communication time is almost negligible compared to the assembly time, but yields to a small time increase for the larger linear system.

Solving the assembled linear system can be done with either a sequential or parallel solver, depending on the size of the system and the available parallel architecture. The sparse linear system resulting from TPSFEM can be solved with a Conjugate-Gradient iterative solver approach [9].

The systems currently generated by HISURF and ADDFIT are dense and symmetric, positive definite in the former case, and semi-definite in the latter case. However, in future refinements of these models, the definiteness property may be lost, for example because of the addition of extra constraints or – in the case of additive models – extending it to a second-order model.

For HISURF and ADDFIT a solver is thus required that will be accurate for any symmetric dense system, and also has good parallel and sequential performance. The former requirement argues for a direct solver with good stability properties; the latter argues for one that exploits symme-

try to require only $\frac{m^3}{3} + O(m^2)$ floating point operations, and that has been shown to have an efficient parallelisation. A direct solver for general symmetric (indefinite) systems based on the diagonal pivoting method [2, 11] meets these requirements.

In the diagonal pivoting method, the decomposition $A = LDL^T$ is performed, where L is an $m \times m$ lower triangular matrix with a unit diagonal, and D is a block diagonal matrix with either 1×1 or 2×2 sub-blocks [11]. The factorisation of A proceeds column by column; in the elimination of column j, three cases arise:

1. Eliminate using a 1×1 pivot from $A_{j,j}$. This corresponds to the definite case, and will be used when $A_{j,j}$ is sufficiently large (compared with $\max(A_{j+1:m,j})$).

2. Eliminate using a 1×1 pivot from $A_{i,i}$, where $i > j$. This corresponds to the semi-definite case; a symmetric interchange with row/columns i and j must be performed.

3. Eliminate using a 2×2 pivot using columns i' and i ($i', i \geq j, i' \neq i$). This case produces a 2×2 sub-block at column j of D. This corresponds to the indefinite case; a symmetric interchange with rows/columns i', i and $j, j + 1$ must be performed. However, columns j and $j + 1$ are eliminated in this case.

The tests used to decide between these cases, and the searches used to select column i (and i'), yield several algorithms based on the method, the most well-known being the variants of the *Bunch-Kaufman* algorithm (see [11] and the references cited within).

It has been recently shown for the *Bunch-Kaufman* algorithm that there is no guarantee that the growth of L is bounded [2]. Variants such as the *bounded Bunch-Kaufman* and *fast Bunch-Parlett* algorithms have been devised which overcomes this problem. The extra accuracy of these methods results from more extensive searching for stable pivot columns i (and i') for cases 2 and 3, with a correspondingly more frequent use of these cases.

For linear systems that are close to definite, such as are likely to be generated by our models, the diagonal pivoting methods permit most columns to be eliminated by case 1, requiring no symmetric interchanges. For a parallel implementation, this is a highly useful property, as even for large matrices the communication startup and volume overheads of symmetric interchange, when the rows and columns come from different nodes, is considerable [22].

Instead of suppressing interchanges, which even if done judiciously may result in the loss of some accuracy [22], high parallel performance can also be achieved with a *block-search* algorithm that searches for suitable pivot columns i and i' from the current storage block [16]. If this search was successful, the symmetric interchanges would require no communication, resulting in no parallel overhead. Such a strategy could be based on the *Duff-Reid* algorithm used for sparse matrices [2, 16], which also has strong guarantees of accuracy.

However, if the search was not successful, an equally stable means of eliminating column j must then be used. We chose the *bounded Bunch-Kaufman* algorithm over the *fast Parlett-Reid* algorithm, as the latter requires sorting of the columns by the size of the diagonal, which would give it higher parallel overheads.

The solving of the linear systems described in Figure 4 took less than one second for the small system and between 20 (sequential) and 11 (parallel) seconds for the large system. On the used cluster architecture even a matrix dimension of 2111 is too small to be efficiently solved on more than just a few computing nodes.

5 Outlook

We are currently working both on the toolbox and on the parallel algorithms. Planned extensions will be more domain independent high-level analysis functions and the inclusion of other parallel data mining algorithms besides predictive modelling, like clustering and association rules.

On the parallel predictive modelling side we plan to add support for data types other than continuous and categorical variables. We hope to include in a first instance support for sets, time series and hierarchical data types.

Acknowledgements

This research was partially supported by the Australian Advanced Computational Systems CRC, and Peter Christen was funded by grants from the *Swiss National Science Foundation* (SNF) and the *Novartis Stiftung, vormals Ciba-Geigy Jubiläums-Stiftung*, Switzerland.

References

[1] D. Aberdeen, J. Baxter and R. Edwards, *98 c/MFlop, Ultra-Large-Scale Neural Network Training on a PIII Cluster*, Gordon Bell award price/performance entry. Submitted to the High-Performance Networking and Computing Conference, Dallas, November 2000.

[2] C. Ashcraft, R.G. Grimes, and J.G. Lewis, *Accurate Symmetric Indefinite Linear Equation Solvers*, Simax, 20(2), 1998.

[3] G. Bell and J.N. Gray, *The revolution yet to happen*, Beyond Calculation (P.J. Denning and R.M. Metcalfe, eds.), Springer Verlag, 1997.

[4] D.M. Beazley and P.S. Lomdahl, *Extensible message passing application development and debugging with Python*, Proceedings 11th International Parallel Processing Symposium, April 1–5, 1997, Geneva, Switzerland, IEEE Computer Society Press, 1997.

[5] D.M. Beazly, *Python Essential Reference*, New Riders, October 1999.

[6] S. Chaudhuri, *Data Mining and Database Systems: Where is the Intersection*, Bulletin of the IEEE Technical Committee on Data Engineering, num. 21, March 1998.

[7] M.-S. Chen, J. Han and P.S. Yu, *Data Mining: An Overview from a Database Perspective*, IEEE Transactions on Knowledge Discovery and Data Engineering, Vol. 8, No. 6, December 1996.

[8] P. Chapman, R. Kerber, J. Clinton, T. Khabaza, T. Reinartz and R. Wirth, *The CRISP-DM Process Model*, Discussion paper, March 1999. `www.crisp.org`

[9] P. Christen, M. Hegland, O.M. Nielsen, S. Roberts, P.E. Strazdins and I. Altas, *Scalable Parallel Algorithms for Surface Fitting and Data Mining*, accepted for the Elsevier Journal of Parallel Computing, special issue on Aspects of Parallel Computing for Linear Systems and Associated Problems, September 2000.

[10] D. Düllmann, *Petabyte databases*. Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-99), ACM Press, July 1999.

[11] G. Golub and C. Van Loan, *Matrix Computations*, John Hopkins University Press, Baltimore, Second edition, 1989.

[12] W. Gropp, E. Lusk and A. Skjellum,*Using MPI – Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, Cambridge, Massachusetts, 1994.

[13] T.J. Hastie and R.J. Tibshirani, *Generalized additive models*, Chapman and Hall, Monographs on statistics and applied probability 43, 1990.

[14] J.M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman,T. Roth and P. Haas, *Interactive Data Analysis: The Control Project*, IEEE Computer, Vol. 32, August 1999.

[15] A.M. Kuchling, *The Python DB-API*, Linux Journal, May 1998.

[16] J.G. Lewis. Private communications, 1999.

[17] W.A. Maniatty and M.J. Zaki, *A Requirements Analysis for Parallel KDD Systems*, IPDPS'2000 Data Mining Workshop, Cancun, Maxico, May 2000.

[18] O.M. Nielsen, P. Christen, M. Hegland and T. Semenova, *A Toolbox Approach to Flexible and Efficient Data Mining*, Accepted for the Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'2001), Hong Kong, April 2001.

[19] S.G. Parker, C.R. Johnson and D.M. Beazley, *Computational Steering Software Systems and Strategies*, IEEE Computational Science & Engineering, 1997.

[20] D. Pyle, *Data Preparation for Data Mining*. Morgan Kaufmann Publishers, Inc., 1999.

[21] P.G. Selfridge, D. Srivastava and L.O. Wilson, *IDEA: Interactive Data Exploration and Analysis*, Proceedings of the ACM SIGMOD International Conference on Management of Data, 1996.

[22] P.E. Strazdins, *Accelerated Methods for Performing the LDLT Decomposition*, Proceedings of CTAC-99: The 9th Biennial Computational Techniques and Applications Conference and Workshops, Canberra, September 1999.

[23] R.J. Yarger, G. Reese and T. King, *MySQL & mSQL*, O'Reilly, July 1999.

[24] M.J. Zaki and C-T. Ho, *Large-Scale Parallel Data Mining*, Lecture Notes in Computer Science and Lecture Notes in Artificial Intelligence, Vol. 1759, Springer-Verlag, 2000.

An Efficient Association Mining Implementation on Cluster of SMPs *

Ruoming Jin Gagan Agrawal
Department of Computer and Information Sciences
University of Delaware, Newark DE 19716
{jrm,agrawal}@cis.udel.edu

Abstract

In this paper, we present our experiences in implementing a hierarchical parallel version of apriori association mining using a middleware. The middleware supports both shared memory and distributed memory parallelization, and enables high I/O performance. Our experimental results on a cluster of SMPs show that: 1) distributed memory parallelization achieves very high efficiency, 2) shared memory parallelization achieves good efficiency if the application is not I/O bound, and 3) the processing time required per data item remains almost the same as the dataset size is increased, establishing that efficient processing of disk resident datasets is possible.

1 Introduction

Association mining is one of the most important data mining tasks. Over the last 6 years, several sequential and parallel algorithms have been developed for association mining. An excellent survey is available from Zaki [23].

Parallel association mining algorithms have been developed for both distributed memory and shared memory architectures. Some recent efforts have also targeted hierarchical systems like cluster of SMP workstations, which have both distributed memory and shared memory parallelism [22]. We believe that two of the important challenges still remaining in developing parallel implementations of data mining tasks are:

1) Developing efficient implementations that process disk resident datasets. The amount of data available for analysis can be huge, and can easily exceed the aggregate main memory available on today's small and medium sized parallel machines.

2) Implementing parallel data mining algorithm without very high programming effort. Currently, developing a parallel implementation that processes disk resident datasets on a cluster of SMPs typically requires programming with the Message Passing Interface (MPI), Posix threads, and file I/O. Thus, developing, debugging, performance tuning, and maintaining a parallel data mining application requires both high expertise and effort.

In our recent work, we have developed a middleware for enabling rapid implementation of parallel data mining applications. This middleware can help exploit parallelism on both shared memory and distributed memory configurations, while allowing efficient processing of disk resident data.

This paper presents a case study on the use of this middleware. We have developed an implementation of apriori association mining on cluster of SMP workstations. The distributed memory parallelization strategy we use is the same as in the well known count distribution technique [1], except that communication is handled by the middleware. Our middleware allows a number of strategies for shared memory parallelization of data intensive reduction operations, including full replication, partial replication, full locking, and fixed locking. Thus, the use of our middleware results in a new family of parallel association mining algorithms for hierarchical systems.

Our experiments on a cluster of SUN SMPs shows that 1) distributed memory parallelization achieves very high efficiency, 2) shared memory parallelization achieves good efficiency if the application is not I/O bound, and 3) the processing time required per data item remains almost the same as the dataset size is increased, establishing that efficient processing of disk resident datasets is possible.

The rest of the paper is organized as follows. We give an overview of our middleware in Section 2. The parallel algorithm we use and its implementation using the middleware is presented in Section 3. The experimental evaluation of our implementation is presented in Section 4. We compare our work with related research efforts in Section 5 and conclude in Section 6.

2 Middleware for Parallel Data Mining Implementations

2.1 Motivation

We have developed a middleware for enabling rapid development of parallel data mining applications. This middleware can help exploit parallelism on both shared memory and distributed memory configurations, while allowing efficient processing of disk resident data.

*This research was supported by NSF CAREER award ACI-9733520, NSF grant CCR-9808522, and NSF grant ACR-9982087. The equipment for this research was purchased under NSF grant EIA-9703088.

Our middleware is based on the observation that parallel versions of several well-known data mining techniques share a relatively similar structure. We have carefully studies parallel versions of apriori association mining [1], bayesian network for classification [9], k-means clustering [13], k-nearest neighbor classifier [12], and artificial neural networks [12]. In each of these methods, parallelization can be done by dividing the data instances (or records or transactions) among the nodes. The computation on each node involves reading the data instances in an arbitrary order, processing each data instance, and performing a *local reduction*. The reduction involves only commutative and associative operations, which means the result is independent of the order in which the data instances are processed. After the local reduction on each node, a *global reduction* is performed. This similarity in the structure can be exploited by the middleware system to execute the data mining tasks efficiently in parallel, starting from a relatively high-level specification of the technique.

2.2 Middleware Interface

The interface exploits the similarity among parallel versions of several data mining techniques, as described in the previous subsection. It assumes that data instances have already been partitioned among the nodes.

The following functions need to be written by the application developer using our middleware. Most of these functions can be easily extracted from a sequential version that processes memory resident datasets.

Initial Processing: Many data mining applications involve an initial processing of the data instances to remove noise or exceptional cases, or modify the format of certain fields, etc. This processing is performed independently on each data instance, and therefore, can be performed in parallel and in an arbitrary order on each processor.

Specifying the Subset of Data to be Processed: In many case, only a subset of the available data needs to be analyzed for a given data mining task. For example, while creating associations rules from customer purchase record at a grocery store, we may be interested in processing records obtained in certain months, or for customers in a certain age groups, etc.

Local Reductions: The data instances owned by a processor and belonging to the subset specified are read. A local reduction function specifies how, after processing one data instance, a *reduction object* (declared by the programmer), is updated. This processing must be independent of the order in which data instances are processed on each processor. The order in which data instances are read from the disks is determined by the runtime system. The reduction object is maintained in the main memory.

Global Reductions: The reduction objects on all processors are combined using a global reduction function.

Iterator: A parallel data mining application comprises of one or more distinct pairs of local and global reduction functions, which may be invoked in an iterative fashion. An iterator function specifies a loop which is initiated after the initial processing and invokes local and global reduction functions.

2.3 Runtime Techniques

We now describe the basic functionality of our middleware system. This system has been developed on top of the Active Data Repository (ADR) developed at University of Maryland [6, 7, 8]. ADR targeted strictly distributed memory parallel machines. We have implemented a framework for runtime parallelization on shared memory machines that allows us to use a cluster of SMP workstations. We are also working on modifying the interface of ADR to make it more suitable for parallel data mining algorithms.

We initially review the basic design and functionality of ADR. Then, we present our producer/consumer framework for runtime parallelization.

Active Data Repository (ADR) [6, 7, 8] is a runtime infrastructure that integrates storage, retrieval and processing of multi-dimensional datasets on a distributed memory parallel machine. ADR runtime support has been developed as a set of modular services implemented in C++. ADR allows customization for application specific processing, while leveraging the commonalities between the applications to provide support for common operations such as memory management, data retrieval, and scheduling of processing across a distributed memory parallel machine. Customization in ADR is achieved through C++ class inheritance. That is, for each of the customizable services, ADR provides a set of C++ base classes with virtual functions that are expected to be implemented by derived classes. Adding an application-specific entry into a modular service requires the definition of a class derived from an ADR base class for that service and providing the appropriate implementations of the virtual functions.

Any task is executed in ADR using two phases: *task planning* and *task execution*. The objective of task planning is to determine a schedule to efficiently process the computation, based on the amount of available resources in the parallel machine. A task plan specifies how parts of the final output are computed. The task execution service manages all the resources in the system and carries out the task plan generated by the task planning service. The primary feature of the task execution service is its ability to integrate data retrieval and processing for a wide variety of applications. This is achieved by pushing processing operations into the storage manager and allowing processing operations to access the buffer used to hold data arriving from disk. As a result, the system avoids one or more levels of copying that would be needed in a layered architecture where the storage manager and the processing belonged to different layers. To further reduce task execution time, the task execution service overlaps I/O, interprocessor communication and processing as much as possible. It does this by maintaining explicit queues for each kind of operation (data retrieval, message sends and receives, data processing) and switches between them as required. A dataset in ADR is partitioned into a set of chunks to achieve high bandwidth data retrieval.

A chunk consists of one or more data items, and is the unit of I/O and communication.

2.4 Runtime Parallelization on SMPs

We now describe the framework we have implemented for efficiently using multiple processors available on each node of a SMP workstation for data mining applications.

The processors available on each node need to perform the following tasks: 1) manage disk operations and file I/O, 2) manage communication with other nodes, 3) execute the *Iterator* loop described earlier, 4) perform runtime scheduling, i.e. assign local reductions on data items being processed by the node to different processors, and 5) perform local reductions.

To perform the above tasks, we use one *producer* thread and one or more *consumer* threads. The producer thread is responsible for the tasks 1, 2, 3, and 4 in the list above. Typically, one consumer thread is scheduled on a single processor, and perform local reductions (task 5) on the data items assigned to it.

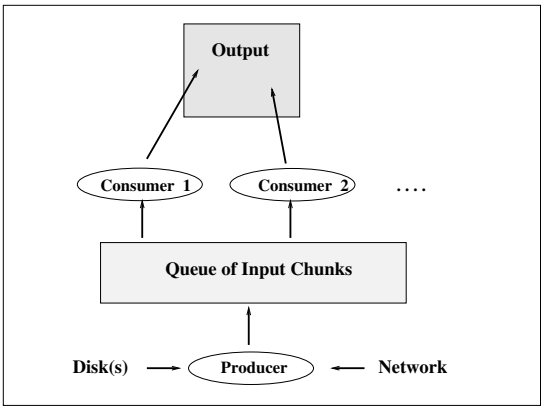

Figure 1. Producer / Consumer Framework for Runtime Parallelization

The producer/consumer framework is shown in Figure 1. As we mentioned earlier in this section, the unit for I/O in ADR is a *chunk*, which is typically one disk block or a small number of disk blocks. We use the same unit for dividing the local reductions among processors on a node. The idea is that chunk size can be chosen to be of sufficiently low granularity to allow effective load balancing at runtime and of sufficiently high granularity to keep the overhead of runtime scheduling acceptable.

In the set of data mining algorithms this middleware is targeting, the local reductions on data items are independent operations, except for race conditions in updating the same reduction object. For example, in the apriori association mining algorithm [1], counts for one or more candidates may be incremented after processing a data item. However, the main complication is that the particular element(s) in the reduction object that need to be modified after processing a data item is not known until after performing the computation associated with the data item.

With this constraint, we have implemented four different approaches for avoiding race conditions as different consumer threads may want to update the same elements in the reduction object. These techniques are: *full locking*, *fixed locking*, *full replication*, and *partial replication*.

Full Locking: One obvious solution to avoiding race conditions is to associate one lock with every element in the reduction object. After processing a data item, the consumer thread needs to acquire the locks associated with all elements in the reduction object it needs to update. If the total number of elements in the reduction object is large, the total number of locks required can be large and can result in significant overheads.

Fixed Locking: To alleviate the overheads associated with the large number of locks required in the full locking scheme, we designed the fixed locking scheme. As the name suggests, a fixed number of locks are used. The number of locks chosen is a parameter to this scheme. If the number of locks is l, then the element i in the reduction object is assigned to the lock $i \ mod \ l$. Clearly, this scheme avoids the overheads associated with supporting a large number of locks in the system. The obvious tradeoff is that as the number of locks is reduced, the probability of one thread having to wait for another one increases.

Full Replication: One simple way of avoiding race conditions is to replicate the reduction object and create one copy for every consumer thread. The copy for each consumer thread needs to be initialized in the beginning. After the local reduction has been performed using all the data items on a particular node, the increments made in all the copies are *merged*. The *global reduction* function specified by the user can be used to perform this merge. After this merge. the global reduction still needs to be performed across the nodes.

Partial Replication: Full replication has the benefit of not requiring any locks and not requiring any thread to wait to acquire a lock. The disadvantages are memory requirements, the need for initializing the elements in the beginning, and performing the merge in the end. To avoid this overhead, we designed the partial replication scheme. Instead of creating copies of the reduction object for each consumer thread, we create a buffer for every consumer thread. The consumer thread stores the updates to elements of the reduction object in this buffer. To keep the memory overhead low, there is a fixed maximum size associated with each buffer. A separate thread works on updating the reduction object using the values from the buffers. Only a single lock is associated with the entire reduction object, so values from only one buffer can be copied into the reduction object at any time.

3 Association Mining: Parallel Algorithm and Implementation

In this section we describe the parallel association mining algorithm we have implemented, and how the middleware interface was used for implementing the algorithm on a cluster of SMPs. We also briefly discuss how a number of other association mining algorithms can be parallelized using our middleware. Initially, we review the well known sequential apriori association mining algorithm [2].

$L_0 = \emptyset$
$C_1 =$ { 1-item subsets of all the transactions }
for $(k = 1; C_k \neq \emptyset\ ; k++)$
  { **support counting** }
  for all transactions $t \in \mathcal{D}$
    for all k-subsets s of t
      if $k \in C_k$
        $s.count++$
  { **candidates generation** }
  $L_k = \{c \in C_k \mid c.count \geq min\ sup\}$
  $C_{k+1} = apriori\_gen(L_k)$
$Answer = \bigcup_k L_k$

Figure 2. Sequential Apriori Association Mining Algorithm

3.1 Sequential Apriori Algorithm

The high level structure of Apriori is shown in Figure 2. The main observation in the Apriori technique is that if an itemset occurs with frequency f, all the subsets of this itemset also occur with at least frequency f. The algorithm employs an iterative approach known as *level-wise* search. During the k^{th} iteration, the algorithm scans the input dataset, and computes the frequent k-itemsets, L_k. There are two stages in the processing of the k^{th} iteration: the *support counting stage* and the *candidate generation stage*. In the first stage, each transaction in the dataset is processed to compute the frequency of the members of the set C_k. In the second stage, the set of frequent k-itemsets L_k is formed by including the k-itemsets in C_k that satisfy the minimum support condition. Then, the candidate (k+1)-itemsets, C_{k+1}, are generated for use in the next iteration.

3.2 Hierarchical Parallelization

We have used the middleware described in the previous section for parallelizing apriori on a cluster of SMP workstations. Our implementation efficiently processes disk resident datasets. We started from a sequential apriori code that processes datasets that fit in main memory. This sequential code was developed by Borgelt [4] and uses a prefix tree to represent candidate itemsets.

The dataset is partitioned between different nodes in the cluster. On each node, the set of transactions is further divided into chunks, which is the unit for processing in our middleware.

The parallelization strategy we use across nodes is the same as the well known count distribution scheme [1], except that we use middleware for handling message passing. We focus on parallelizing support counting only, as it is the most time consuming part of the algorithm. The support counting stage in every iteration is completed by two phases: the *local count* phase and the *global count* phase. In the local count stage, every node computes the count for each candidate among the set of transactions locally owned. In the global count stage, the nodes exchange the local counts with other nodes, to compute the final counts for candidates.

The middleware's producer/consumer framework is used for parallelization within the node. The use of four mechanisms implemented in the middleware for avoiding race conditions while updating candidate counts leads to a family of new parallel apriori algorithms for hierarchical systems. We use up to m threads on each node, where $m = n + 1$, and n is the number of processors on each node. One thread, referred to as the producer thread, is used to coordinate disk operations, communication with other nodes, and performing global reductions. Other threads, referred to as consumer threads, perform local reductions.

The shared memory parallelization we perform has some similarities with the Common Candidate Partitioned Database (CCPD) scheme developed by Zaki *et al.* [24], in the sense that there is only one itemset tree stored in the main memory. However, there are two main differences in our approach. First, the use of middleware and a producer thread allows asynchronous and efficient I/O, and therefore, good performance on disk resident datasets. Second, depending upon the scheme used for avoiding race conditions, the candidate counts may be fully or partially replicated across processors, enabling higher parallelism.

The major functions used in middleware specification are shown in Figure 3. The function `LocalReduc` just invokes the function `prefixtree.count` from the sequential implementation. After the local reduction, the reduction object on each processor (counts of all candidates) are broadcasted to all other processors. The processing in the `GlobalReduc` function is very simple. In the function shown in the figure, counts of all candidates received from another processor are combined with the local counts. This is repeated for the buffer received from each processor. The `Iterator` function invokes local and global reductions and adds candidates to the prefix tree.

3.3 Discussion

The salient and novel charactestics of our parallel algorithm and its implementation are as follows. First, we have used a middleware for specifying the algorithm. The middleware specification does not involve use of MPI, Posix threads, or file I/O. We believe that our implementation is the first hierarchical parallel implementation of apriori on disk resident datasets. The runtime techniques implemented in the middleware perform aggressive I/O optimization. Finally,

```
LocalReduc(Chunk p, int k)
{
   for (each transaction t in p)
   {
      prefixtree.count(c,t,k);
   }
}

GlobalReduc(CountVector c, MessageReceiveBuffer buffer)
{
   combine(c,buffer);
}
```

```
Iterator()
{
   for (k = 1 ; not end_condition() ; k++)
   {
      for (each chunk p)
         LocalReduction(p,k);
      GlobalCombination(c,buffer);
      prefixtree.expand();
   }
}
Bool end_condition()
{
   return prefixtree.newcandidate == 0;
}
```

Figure 3. Main functions used in Middleware Specification

by using the four techniques implemented in the middleware for avoiding race conditions while updating candidate counts, we have a new family of hierarchical parallel association mining algorithms.

Our basic scheme and the middleware can also be used for parallelization of a number of other association mining algorithms and variations of apriori, including SEAR [14], DHP [15], Partition [19], and DIC [5]. These algorithm differ from the apriori algorithm in the data structure used for representing candidate itemsets, candidate space pruning, or in reducing passes over the set of transactions. The same distributed memory and shared memory parallelization strategies can be applied on these algorithms.

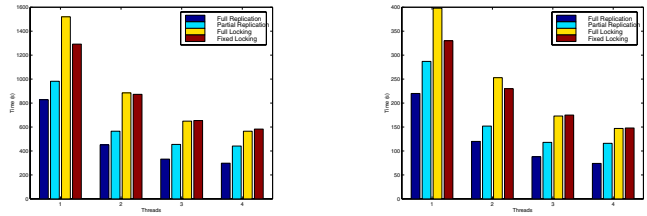

Figure 4. Comparing the Four Strategies for SMP Parallelization (800 MB dataset): 2 nodes (left), 8 nodes (right)

4 Experimental Results

In this section, we evaluate our implementation of apriori association mining.

4.1 Experimental Platform

The experiments were conducted on a cluster of SMP workstations. We used 8 Sun Microsystem Ultra Enterprise 450's, each of which has 4 250MHz Ultra-II processors. Each node has 1 GB of main memory which is 4-way interleaved. Each of the node have a 4 GB system disk and a 18 GB data disk. The data disks are Seagate-ST318275LC with 7200 rotations per minute and 6.9 milli-second seek time. The nodes are connected by a Myrinet switch with model number M2M-OCT-SW8. We believe that our cluster represents a common parallel processing configuration using off-the-shelf nodes and network.

4.2 Evaluating Shared Memory Parallelization Techniques

We designed a detailed experiment to both evaluate the overall efficiency of our producer/consumer framework and compare the four techniques for runtime parallelization. We used a dataset with 8 million transactions, each with 20 items (on the average). The total number of distinct items in the dataset is 1000. The total size of the dataset is 800 MB. Because our focus in this experiment is on evaluating SMP parallelization techniques, we choose a dataset that could fit in the main memory of a single node.

The association mining implementation developed on top of our middleware was executed using each of the four techniques, on 1, 2, 4, and 8 nodes of the cluster and using 1, 2, 3 or 4 consumer threads per node. The performance comparison on 2 nodes is shown in the left of Figure 4 and the performance comparison on 8 nodes is shown in the right of Figure 4. The sequential version took 1639 seconds. The threads per node listed in the Figures are consumer threads performing actual computations. In each case, there is a separate producer thread per node.

On 2 nodes and 1 thread per node, the speedups are 1.97 with full replication, 1.67 with partial replication, 1.07 with full locking, and 1.27 with fixed locking. The performance of full replication shows that distributed memory parallelization is working very well and resulting in almost perfect speedups. The performance of the other three schemes shows that there are significant overheads with shared memory parallelization using these schemes, even though only 1 thread is used on every node. The overhead of full locking is very high because of the large number of locks that need to be used. The performance of fixed locking (using 1024 locks) is significantly better, though not comparable with the performance of full replication or partial replica-

tion. With only 1 thread, the fixed locking scheme reduces the overhead associated with supporting a large number of locks in the system, but does not loose any parallelism.

On 2 nodes and 2 threads per node, the speedups are 3.63 with full replication, 2.9 with partial replication, 1.85 with full locking, and 1.87 with fixed locking. All the versions have significant speedups compared to the 1 thread versions, using the same scheme, on 2 nodes. The relative speedups compared to 1 thread versions are 1.83 with full replication, 1.73 with partial replication, 1.72 with full locking, and 1.47 with fixed locking. The performance of full replication version with 2 threads shows that the dynamic work assignment as part of the producer/consumer framework is working without any significant overheads. The partial replication version incurs some overheads for updating the main buffer and looses some performance. Its performance is still significantly better than the two locking versions. One main observation from the results with 2 threads per node is that the performance of fixed locking and full locking is very similar. Though fixed locking has lower overhead for supporting the locks, it also results in some loss of parallelism.

On 2 nodes and 3 threads per node, the speedups are 4.94 with full replication, 3.6 with partial replication, 2.52 with full locking, and 2.5 with fixed locking. The trends are very similar to the 2 threads per node case. All the schemes are successfully exploiting the extra thread to improve performance. Fixed locking actually performs worse than full locking in this case, though the difference is less than 1%.

On 2 nodes and 4 threads per node, the speedups are 5.5 with full replication, 3.71 with partial replication, 2.9 with full locking, and 2.81 with fixed locking. The relative performance improve because of adding an extra thread per node is relatively small for each of the versions. This is because each node has 1 producer thread, in addition to the 1, 2, 3 or 4 consumer threads. When 4 consumer threads are used, the total number of threads on 4 processors is 5, resulting in some contention. The fourth consumer thread still results in some additional speedups, which shows that the producer thread is not active all the time.

The results from comparing the 4 schemes on 8 nodes, with 1, 2, 3, and 4 threads per node, are shown in the right of Figure 4, and are similar to the 2 node results.

4.3 Evaluating I/O Scalability

One of the obvious questions with any data mining implementation is, "How does the performance scale with the increase in the size of the dataset ?". Particularly interesting is the time required per data item or transaction when the dataset is memory resident and when the dataset becomes disk resident.

We designed an experiment to evaluate this for our implementation of association mining. The main challenge in evaluating the effect of disk resident data on the performance is in keeping the amount of computation per transaction unchanged as the total number of transactions is increased. The amount of computation per transaction depends upon the number of candidates. To keep the number

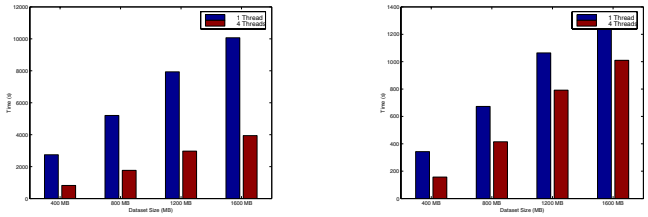

Figure 5. I/O Scalability of Association Mining Algorithm: 0.5% support level (left), 1% support level (right)

of candidates unchanged as the number of transactions is increased, we created 4 different datasets as follows. We initially created a 400 MB dataset, comprising 6 million transactions, with an average of 15 items per transaction. We then duplicated the data to create a 800 MB dataset, repeated it three times to create a 1.2 GB dataset, and finally, repeated it four times to create a 1.6 GB dataset. If the same support percentage is used, the number of candidates considered during any iteration will be the same for these 4 datasets. The first and the second datasets are memory resident while the third and the fourth dataset exceed the main memory limit on 1 node.

Since we were only interested in evaluating the I/O performance in this experiment, we use a single node. We used 1 or 4 threads on this node. We used two different support levels, 0.5% and 1%, on the same 4 datasets. The total number of candidates that have to be considered with the support level of 0.5% is very large, and 13 iterations of the outer-loop in the apriori association mining algorithm are required. With the support level of 1%, only 4 iterations of the outer-loop are required.

The performance on the four datasets, with 1 and 4 threads on 1 node and the support level of 0.5%,is shown in the left of the Figure 5. With 1 thread, the ratio of the time spent on the 800 MB, 1.2 GB, and 1.6 GB dataset to the time spent on the 400 MB dataset is 1.9, 2.9, and 3.67, respectively. This shows that the time required per transaction is not increasing as we move from memory resident dataset to disk resident dataset. Some of the computation in the code is independent of the number of datasets processed, which explains why the processing time increases by a factor less than linear on the size of the dataset.

With 4 threads, the ratio of the time spent on the 800 MB, 1.2 GB, and 1.6 GB dataset to the time spent on the 400 MB dataset is 2.15, 3.62, and 4.80, respectively. As more I/O is required, the producer thread takes more cycles and slows down the computation when 4 threads are used.

The performance with 1 and 4 threads on 1 node and the support level of 1.0% is shown in the right of Figure 5. With 1 thread, the ratio of the time spent on the 800 MB, 1.2 GB, and 1.6 GB dataset to the time spent on the 400 MB dataset is 1.96, 3.10, and 3.93, respectively. With 4 threads,

the ratio of the time spent on 800 MB, 1.2 GB, and 1.6 GB dataset to the time spent on the 400 MB dataset is 2.64, 5.04, and 6.43, respectively. As the code becomes more I/O dominated with the use of 1.0% support level, the processing time increases by more than a linear factor as the dataset size increases. However, the rate of increase is still within reasonable levels, and we believe that it shows that our system gives good performance on disk resident datasets.

4.4 Overall Performance

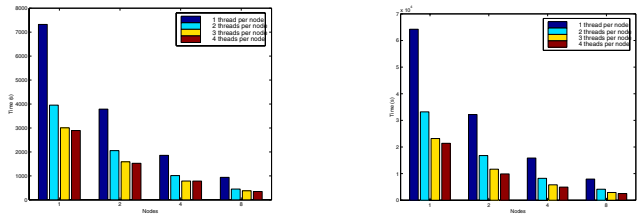

Figure 6. Performance of Apriori Association Mining: `2GB` dataset (left), `8GB` dataset (right)

In this subsection, we focus on the overall performance achieved on the implementation of apriori association mining. We have used two large datasets for this purpose. The first dataset, which is referred to as the `2GB` dataset in our presentation, has 32 million transactions, each with (on the average) 20 items, and with 1000 distinct items. The total size of the dataset is 2.8 GB. Thus, this dataset does not fit in the main memory when the code is executed on 1 or 2 nodes. The second is referred to as the `8GB` dataset. This dataset has 64 million transactions, with an average of 30 items per transaction. The total number of distinct items is 1000. The support and confidence levels used in our experiments are 1% and 90%, respectively, for both the datasets. Since we wanted to see the best performance than can achieved using our system, we have used the full replication scheme in all the experiments presented in this subsection.

The performance on the `2GB` dataset is shown in the left of Figure 6. Results from 1, 2, 4 and 8 nodes and 1, 2, 3, and 4 threads per node are presented. This dataset results in 8 iterations of the outer-loop in the apriori association mining algorithm. The number of candidates whose support is counted is 1,000, 278,735, 43,144, 4,354, 8,373, 949, 35, and 1, for the first through eighth iterations, respectively. Therefore, each node needs to send a total of nearly 1.3 MB of data (broken over 8 messages) to every other node, and needs to receive the same amount of data from every other node.

The middleware achieves high parallel efficiency for both distributed memory and shared memory parallelization. Using only 1 thread per node, the speedups on 2, 4, and 8 nodes are 1.93, 3.94, and 7.8, respectively. Note that the dataset owned by each node becomes main memory resident in going from 2 to 4 nodes. This results in a superlinear speedup in going from 2 to 4 nodes.

The shared memory parallelism is also exploited well on up to 3 threads per node. Using 3 threads per node, the speedups on 1, 2, 4, and 8 nodes are 2.43, 4.61, 9.31, and 19.32, respectively. Because of the producer thread, the fourth consumer thread does not result in significantly better performance. Using 4 threads per node, the speedups on 1, 2, 4, and 8 nodes are 2.53, 4.80, 9.35, and 21.04, respectively. The parallel efficiency in using 8 nodes and 3 threads per node is 81%, and the parallel efficiency in using 8 nodes and 4 threads per node is 66%.

The performance on the `8GB` dataset is presented in the right of Figure 6. This dataset resulted in 9 iterations of the outer-loop of the apriori association mining algorithm. The number of candidate whose support is counted during these iterations is 1,000, 344,912, 858,982, 25,801, 22,357, 14,354, 6,257, 55, and 1, for the first through ninth iterations, respectively. Each node has to broadcast and receive 5.1 MB of data (broken over 9 messages) during the course of the execution.

Again, high distributed memory and shared memory parallel efficiency is achieved. With the use of 1 thread per node, the speedups on 2,4, and 8 nodes are 1.99, 4.05, and 8.07, respectively. Note that for this dataset, the data is not memory resident even on 8 nodes. However, as the data is distributed over multiple nodes, the amount of I/O needed on each node reduces, and helps achieve high speedups.

Use of up to 3 threads per node results in almost linear performance improvements. With 3 threads per node, the speedups on 1, 2, 4, and 8 nodes are 2.77, 5.50, 11.08, and 21.98, respectively. The parallel efficiency on 8 nodes with 3 thread per node is 91.6%. With 4 threads per node, the speedups on 1, 2, 4, and 8 nodes are 3.0, 6.5, 13.03, and 25.50, respectively. As the I/O requirements per node decrease, the producer thread consumes fewer cycles, resulting in more substantial performance gains with the use of the fourth consumer thread.

5 Related Work

We now compare our work with related research efforts.

Parallelization of association mining techniques is a well studied area [1, 10, 11, 16, 17, 18, 20, 24, 22]. Our work is unique in considering hierarchical parallelism on disk resident datasets, and using a middleware to implement the algorithm without low-level parallel programming. Our method for shared memory parallelization is significantly different from the existing parallel association mining algorithms on shared memory and hierarchical systems [10, 17, 18, 24, 22]. Through the use of our middleware, we combine task and data parallelism, and exploit four new techniques for avoiding race conditions while updating candidate counts.

One effort somewhat similar to our work is from Becuzzi *et al.* [3]. They use a structured parallel programming environment PQE2000/SkIE for developing parallel implementation of data mining algorithms. Our work is distinct at least two important ways. First, they only target distributed

memory parallelism (while they report results on an SMP machine, it is using MPI). Second, I/O is handled explicitly by the programmers in their approach.

Several runtime support libraries and file systems have been developed to support efficient I/O in a parallel environment, most noticeable among these is the PASSION library designed by Alok Choudhary's group [21]. They usually provide a collective I/O interface, in which all processing nodes cooperate to make a single large I/O request. With these collective I/O interfaces, the I/O requests still need to be inserted by the programmers, and data processing usually cannot begin until the entire collective I/O operation completes. The middleware we have presented is significantly different, because the computation is an integrated part of the specification. It is specifically targeted towards the kind of computations arising in data mining algorithms and data intensive reduction operations, whereas I/O libraries like PASSION are more general.

Our middleware system has been developed out the Active Data Repository (ADR) designed by Joel Saltz's group at University of Maryland [6, 7]. ADR could only be used on a cluster of single-processor workstations, and is not tailored for data mining techniques. Our middleware has been tailored for data mining, and runs on a cluster of SMPs.

6 Conclusions

This paper has presented a case study in using our middleware for developing a parallel implementation of apriori association mining. The salient features of our implementation are:

1) Our implementation exploits both shared memory and distributed memory parallelism. By using the four techniques available for shared memory parallelization in our middleware, we have created a new family of parallel association mining algorithms for hierarchical systems.

2) The parallel implementation can process disk resident datasets, thus enabling data mining on large and realistic datasets.

3) Experimental results have shown that i) distributed memory parallelization achieves very high efficiency, ii) shared memory parallelization achieves good efficiency if the application is not I/O bound, and iii) the processing time required per data item remains almost the same as the dataset size is increased, establishing that efficient processing of disk resident datasets is possible.

Acknowledgments

This work builds on top of Active Data Repository (ADR) designed at the University of Maryland by the CHAOS group, lead by Joel Saltz and Alan Sussman. We particularly thank Chialin Chang and Renato Ferreira for helping us in getting started with ADR, and for their help with experiments during the course of this research. We also thank Srini Parthasarathy for giving us many pointers to data mining literature during the initial phases of this research.

References

[1] R. Agrawal and J. Shafer. Parallel mining of association rules. *IEEE Trans. on Knowledge and Data Engg.*, 8(6):962–969, December 1996.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 1994 Int. conf. Very Large DataBases (VLDB'94)*, pages 487–499, Santiago,Chile, September 1994.

[3] P. Becuzzi, M. Coppola, and M. Vanneschi. Mining of association rules in very large databases: A structured parallel approach. In *Proceedings of Europar-99, Lecture Notes in Computer Science (LNCS) Volume 1685*, pages 1441 – 1450. Springer Verlag, August 1999.

[4] Christan Borgelt. Apriori. http://fuzzy.cs.Uni-Magdeburg.de/ borgelt/Software. Version 1.8.

[5] S. Brin, R. Motwani, J. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *ACM SIGMOD Conf. Management of Data*, May 1997.

[6] C. Chang, A. Acharya, A. Sussman, and J. Saltz. T2: A customizable parallel database for multi-dimensional data. *ACM SIGMOD Record*, 27(1):58–66, March 1998.

[7] Chialin Chang, Renato Ferreira, Alan Sussman, and Joel Saltz. Infrastructure for building parallel database systems for multi-dimensional data. In *Proceedings of the Second Merged IPPS/SPDP (13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing)*. IEEE Computer Society Press, April 1999.

[8] Chialin Chang, Tahsin Kurc, Alan Sussman, and Joel Saltz. Query planning for range queries with user-defined aggregation on multi-dimensional scientific datasets. Technical Report CS-TR-3996 and UMIACS-TR-99-15, University of Maryland, Department of Computer Science and UMIACS, February 1999.

[9] P. Cheeseman and J. Stutz. Bayesian classification (autoclass): Theory and practice. In *Advanced in Knowledge Discovery and Data Mining*, pages 61 – 83. AAAI Press / MIT Press, 1996.

[10] D. Cheung, J. Han, V. Ng, A. Fu, and Y.Fu. A fast distributed algorithm for mining association rules. In *4th Intl. Conf. Parallel and Distributed Info. Systems*, December 1996.

[11] E-H. Han, G. Karypis, and V. Kumar. Scalable parallel datamining for association rules. *IEEE Transactions on Data and Knowledge Engineering*, 12(3), May / June 2000.

[12] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2000.

[13] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.

[14] A. Mueller95. Fast sequential and parallel algorithms for association rule mining: A comparison. Technical Report CS-TR-3515, University of Maryland, College Park, August 1995.

[15] J. S. Park, M. Chen, and P. S. Yu. An effecitive hash based algorithm for mining association rules. In *ACM SIGMOD Intl. Conf. Management of Data*, May 1995.

[16] J. S. Park, M. Chen, and P. S. Yu. Efficient parallel data mining for association rules. In *ACM Intl. Conf. Infomration and Knowledge Management*, November 1995.

[17] Srinivasan Parthasarathy, Mohammed Zaki, and Wei Li. Memory placement techniques for parallel association mining. In *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining (KDD)*, August 1998.

[18] Srinivasan Parthasarathy, Mohammed Zaki, Mitsunori Ogihara, and Wei Li. Parallel data mining for association rules on shared-memory systems. *Knowledge and Information Systems*, 2000. To appear.

[19] A. Savasere, E. Omiecinski, and S.Navathe. An efficient algorithm for mining association rules in large databases. In *21th VLDB Conf.*, 1995.

[20] T. Shintani and M. Kitsuregawa. Hash based parallel algorithms for mining association rules. In *4th Intl. Conf. Parallel and Distributed Info. Systems*, December 1996.

[21] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kutipudi. Passion: Optimized I/O for parallel applications. *IEEE Computer*, 29(6):70–78, June 1996.

[22] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *3rd Intl. Conf. on Knowledge Discovery and Data Mining.*, August 1997.

[23] M.J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency (Special Issue on Data Mining)*, 1999.

[24] M.J. Zaki, M. Ogihara, S. Parthasarathy, and W.Li. Parallel data mining for association rules on shared-memory multi-processors.esso. In *Supercomputing'96*, November 1996.

Implementation and Performance Evaluation of Dynamic Scheduling for Parallel Decision Tree Generation

Kazuto Kubota, Akihiko Nakase and Shigeru Oyanagi
Parallel Application TOSHIBA Laboratory, Real World Computing Partnership
R&D Center, Toshiba Corp. Toshiba-cho 1, Komukai, Saiwai-ku,
Kawasaki-shi, Kanagawa-ken, Japan, 212-8582
{kazuto, nakase, oyanagi}@isl.rdc.toshiba.co.jp

Abstract

This paper proposes a parallel data-mining algorithm and its implementation on a PC cluster. The decision tree is a widely used data-mining algorithm for classifying records in a database. Simple parallelization of decision tree generation is not efficient because of the load imbalance caused by the form of the generated tree. The SPRINT algorithm solves this problem by grouping a set of nodes in the same level of the tree and balancing the load; however, frequent disk access is required when the data size exceeds the memory size. We propose an improved parallel algorithm of SPRINT by incorporating a dynamic scheduling. Dynamic scheduling is effective in reducing the amount of disk access for storing intermediate results; however, it may cause imbalance in data distribution on PEs (Processing Elements). We solved this problem by incorporating data redistribution. The evaluation result shows that our method realizes an improvement in speed of 3.5 times, for the best case, and equal performance even in the worst case, compared with SPRINT. We also discuss how further performance enhancement may be possible by improving the communication performance.

1. Introduction

Data-mining tools have been used for decision support in a wide range of business fields. According to the demand for more speed, and the explosive growth in data size, an efficient data-mining system which can rapidly deal with huge amounts of data, is desired. In order to realize an efficient data-mining system, parallel processing is a promising approach. Recent remarkable advances in PCs and deployment of storage technology, enable the PC cluster to be a cost effective solution for the parallel processing platform. Much research has already been carried out on PC clusters in scientific application fields. Since data-mining is a typical example of high performance computing in the business application field, development of a parallel data-mining system on the PC cluster is an important research target.

This paper describes parallel implementation of a decision tree generation algorithm on the PC cluster. The decision tree is a typical data-mining algorithm which is widely used to classify database records. It is superior to the other classification algorithms with respect to its accuracy and execution speed.

Simple parallelization of decision tree generation is not efficient because of the load imbalance caused by the form of the generated tree. The SPRINT algorithm proposed by Shafer et al.[10] solves this problem by grouping a set of nodes in the same level of the tree. The grouping of nodes to be processed is effective for balancing the load and decreasing the barrier synchronization count. The SPRINT algorithm works efficiently when the data size of the grouped nodes can be stored in memory; however, frequent disk access is required when the data size exceeds the memory size.

In this paper, we propose an improved parallel algorithm of SPRINT by incorporating dynamic scheduling. Dynamic scheduling is defined to find an optimal set of nodes to be grouped together in consideration of the memory size. Dynamic scheduling is effective in reducing the amount of disk access for storing intermediate results; however, it may cause imbalance in data distribution on PEs. We solved this problem by incorporating data redistribution.

We developed this parallel algorithm on the PC cluster, and evaluated performance using benchmark data. The results show that our method realizes an improvement in speed of 3.5 times, in the best case, and equal performance even in the worst case, compared with SPRINT. We also discuss how further performance enhancement may be possible by improving the communication performance.

This paper is organized as follows. Section 2 describes the basic algorithm for decision tree generation. Section 3

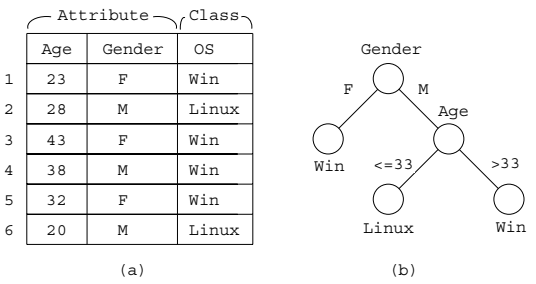

Figure 1. Training set and its decision tree. (a) Training set. (b) Decision Tree.

explains the SPRINT algorithm. Section 4 describes the introduction of dynamic scheduling to SPRINT together with the redistribution technique. The PC cluster used for the experiment is shown in Section 5. Section 6 shows the experimental results, and they are discussed in section 7. Section 8 describes related works, and section 9 is the conclusion.

2. Decision tree generation algorithm

2.1. Decision tree

Decision tree generation is a typical method for classification used in data-mining. A set of records called a training set is given as an input (Fig.1(a)). One record consists of multiple attributes and one class. An attribute may have a discrete value (Gender) or continuous value (Age). A decision tree is a tree which determines the class of a record from its attribute value. Fig.1(b) shows the decision tree generated from the training set of Fig.1(a).

A decision tree is used to classify a test data set which does not have a class value. A decision tree consists of nodes and leaves. Each internal node has a condition which tests an attribute value of a record. A leaf is labeled with a class name. By applying a record to the decision tree from a root node to a leaf node, the class of the record is determined at the leaf node.

2.2. Basic algorithm

The fundamental algorithm of the decision tree generation is shown in Fig.2. This is based on the divide and conquer method. In node generation (`FormTree`), the evaluation values of all partitioning candidates are calculated (`EvalAtt`). When an attribute takes a category value, candidates are the division of node data into two or more subsets by its categories. When an attribute takes a continuous value, every possible division into two subsets are candidates. The candidate which takes the greatest evaluation

```
FormTree( data ) /* node generation */
{
  EvalAtt( data ); /* evaluation */
  DivData( data ); /* data division */

  for each sub data i
    FormTree( subdata[i] );
}
```

Figure 2. Decision tree generation algorithm.

value is selected, and the node data is divided(`DivData`). `FormTree` is applied to the divided data recursively. Many techniques have been proposed for calculating evaluation values, such as the information entropy based method [9], and the gini index based method [2]. In any case, the evaluation value is calculated from the class histogram (the counts of Win and Linux in Fig.1(a)).

3. SPRINT

SPRINT is the decision tree generation algorithm developed by Shafer et al., which is designed for parallel processing. Large-scale data over the memory size can be treated. The performance on the parallel computer SP2 is shown in [10]. Here, its sequential algorithm and parallel algorithm are explained. In this article, the explanation of the algorithm is slightly different from the original paper for the sake of introducing the dynamic scheduling in section 4.1. Refer to [10] for the details of SPRINT.

3.1. Sequential SPRINT algorithm

The outline of the sequential algorithm is shown below.

- **Step 1.** Training set data is read from a file. An attribute list for every attribute is generated from all records. The structure of the attribute list consists of the value (`Val`) of an attribute, a class (`Class`), and a record number (`Id`). Fig.3(a) shows the attribute list generated from the attribute (`Age`) of Fig.1(a).

- **Step 2.** The attribute list which corresponds to a continuous value is sorted by its value (Fig.3(b)).

- **Step 3.** All the nodes at the same depth of the tree are grouped, and the processes from step 3-1 to step 3-5 are executed for each node.

  - **Step 3-1.** Evaluation values are calculated for all the possible partitioning. In SPRINT, only binary division is taken.

Age	Class	Id
23	Win	1
28	Linux	2
43	Win	3
38	Win	4
32	Win	5
20	Linux	6

(a)

Age	Class	Id
20	Linux	6
23	Win	1
28	Linux	2
32	Win	5
38	Win	4
43	Win	3

(b)

Figure 3. Attribute lists of Age of Fig.1(a). (a) Before presorting. (b) After presorting.

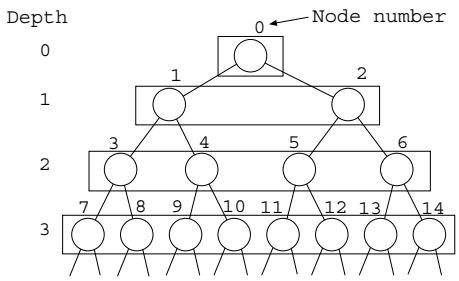

Figure 4. SPRINT and its node grouping.

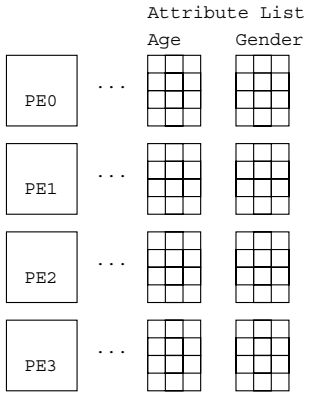

Figure 5. Data placement in parallel SPRINT.

  - **Step 3-2.** Division with the maximum evaluation value in step 3-1 is chosen.
  - **Step 3-3.** In order to divide the node data by the division in step 3-2, a data structure called the probe structure is generated. It is a hash table of record numbers which correspond to one of the divided data.
  - **Step 3-4.** Two new child nodes are created, and each record of the attribute list is assigned to one of the child nodes using the probe structure.
  - **Step 3-5.** If all the records in a child node have the same class, the node becomes a leaf. A class value is attached to the leaf, and records are deleted from attribute lists. If an attribute list is empty, go to step 5.

- **Step 4.** Increase the depth of the tree and return to step 3.
- **Step 5.** End.

The features of the SPRINT algorithm are as follows.

- Sort is executed only once for attribute lists with a continuous value at preprocessing. This enables execution of large-scale data which exceeds memory size, within practical time.
- The nodes of the same depth are grouped and calculated simultaneously.
- The form of any generated decision tree is restricted to a binary tree.

In a decision tree generation program such as C4.5[9], sort is necessary for calculating the evaluation value of each continuous attribute on each node. Since an attribute list keeps the sorted order of continuous attribute in SPRINT, sorting is not necessary.

The grouping of nodes in SPRINT is shown in Fig.4. In this figure, the nodes surrounded by the rectangles are grouped together. This grouping balances the processing load of each PE at parallel processing, and reduces the counts of barrier synchronizations.

3.2. Parallel SPRINT Algorithm

In parallel SPRINT, each attribute list is divided and distributed to each PE (Fig.5). Like a sequential SPRINT, sorting the attribute list of continuous values is performed first, and the calculation of the evaluation value and data partitioning using a probe structure are repeated. In the evaluation of a continuous attribute, a class histogram is calculated in each PE, and exchanged between each one prior to the execution. This creates an independent calculation in each PE. In the evaluation of a discrete attribute, class histograms for each category value are calculated in each PE, and are summarized prior to the execution. A probe structure is generated by merging the local probe structures generated in each PE. A data division can be independently performed at each PE based on a probe structure.

4. Implementation of new parallel decision tree generation algorithm

We propose a new method to incorporate dynamic scheduling into SPRINT. We first incorporate dynamic scheduling with a sequential SPRINT algorithm, and then describe its parallelization. Next, we explain the data redistribution method.

4.1. Dynamic scheduling

In Fig.4, the nodes with the same depth are grouped and calculated simultaneously. However, their calculation requires enough memory size to handle all the data contained in these nodes. When memory size is insufficient, intermediate data should be stored on a disk. If the data size of a root node exceeds the memory size, the processing of nodes in successive depths also exceeds the same memory size. Hence, disk access occurs frequently. In order to decrease disk access, it is necessary to group nodes in consideration of the memory size. We call this technique, dynamic scheduling. Two steps of the SPRINT algorithm are changed to incorporate dynamic scheduling. Sentences in bold represent the changed part.

- **Step 3-4.** Two new child nodes are created, and each record of the attribute list is assigned to one of the child nodes using a probe structure. **From the record count contained in each child node, the memory size required for the processing is calculated. When this size is less than the memory size of a platform, the node is stored in the node queue, and its records are deleted from the attribute lists.**

- **Step 5. A set of nodes that satisfy the memory size condition are fetched from the node queue, and are processed by in-memory execution**.

Fig.6 shows the processing of dynamic scheduling. In this figure, processing of nodes surrounded by the white rectangle requires disk access, and processing of nodes surrounded by the hatched rectangle can be performed in memory. For example, node 3 and node 6 at depth 2 are stored in the node queue, and the processing of node 4 and node 5 are executed with disk access. In the node processing of depth 3, the nodes of 9-12 can be processed in memory. Hence, they are stored in the node queue to be processed later. In the original SPRINT algorithm, all the node processings required disk access. However, this modification can reduce the execution, with disk access at five nodes.

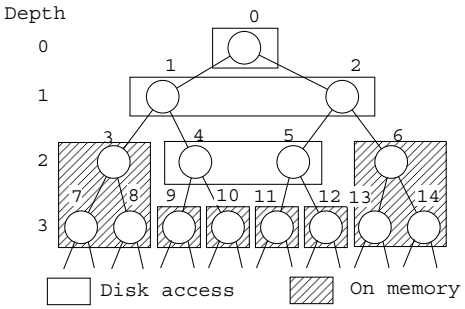

Figure 6. Node grouping with dynamic scheduling.

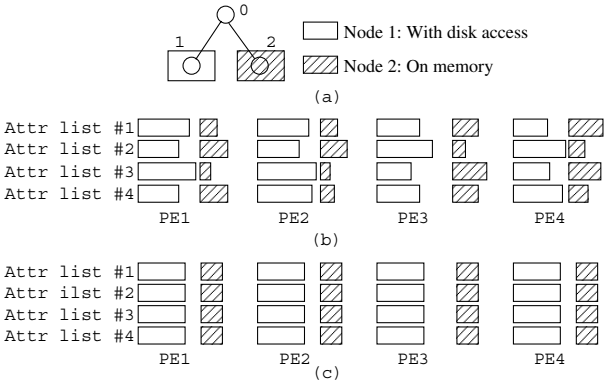

Figure 7. Data distribution in each node. (a) Node 1 and 2 are generated from node 0. (b) Data distribution before redistribution. (c) Data distribution after redistribution.

4.2. Applying the dynamic scheduling to parallel SPRINT

Dynamic scheduling can be similarly applied to the parallel SPRINT. A node can be stored in a node queue when its data size is smaller than the sum of the memory size of all the PEs. When dynamic scheduling is used in the case of parallel processing, the data distribution among PEs becomes unbalanced as the calculation proceeds. Fig.7 shows the data distribution when two nodes are generated after processing of a root node. This data has four attribute lists. Data of the root node is equally distributed to each PE. We assume that node 1 cannot be processed in memory, but node 2 can be processed in memory (Fig.7(a)). Fig.7(b) shows the distribution of the attribute list of node 1 and node 2 after root node division. The white rectangle corresponds to node 1 and the hatched rectangle corresponds to node 2. Note that the size of each attribute list on each PE

differs each from the other. When a dynamic scheduling is performed, processing of node 1 is continued, and node 2 is stored in the node queue to be processed later. In both cases, data distribution is not well balanced; hence, efficient parallel execution cannot be expected. Moreover, the processing of node 2 may not be executed in memory, if the number of records allocated on a PE is large. In order to avoid this, it is necessary to redistribute the data of each node evenly (Fig.7(c)). This technique is called redistribution of data. Redistribution enables efficient parallel processing owing to the evenly balanced load, but the merit of the redistribution will be lost when the time for data movement between the PEs exceeds the time caused by the data imbalance. Hence, we determine a threshold value, and when the largest record number of a node exceeds this value, we perform the redistribution. The redistribution coefficient is defined as the rate of threshold value to the means of the record number of each PE. The threshold value is the multiplication of the means of the record number by the coefficient. Redistribution is performed for each attribute list.

The nodes stored in the node queue can be processed in any order. Each node may be processed individually or may be combined into one task. The number of PEs for node processing can be freely decided in the range within which processing can be performed in memory. Here, we decide that as many nodes as possible are combined into a task to be processed by all PEs under the condition of in-memory processing. This strategy aims to avoid the overhead increase when the node with a small number of data records is individually processed by all PEs.

4.3. Data structures and implementation

Although the decision tree generated by SPRINT is a binary tree, trees with three or more branches are feasible, in general. In order to generate an n-ary tree, we use a new data structure called a translation table, instead of a probe structure. The translation table is the array of node numbers, and its index shows a record number. There are two methods for placing the translation table on the PEs. One is to divide the translation table into the number of PEs, and distribute them to all PEs. The other is to copy the whole translation table to all PEs. Currently, we use the latter method. In table generation, a partial table is generated in each PE and a whole table is generated combining the partial tables. If the size of the training set becomes large, a part of the translation table and/or attribute list should be stored on a disk. To avoid the decrease in performance in this situation, the structure of an attribute list was changed from Fig.3 to Fig.8. Node and Flag are added, and Node expresses the node number to which the record belongs.

Tree generation using a translation table is explained in Fig.9. Fig.9(a) shows a training set which has eight records

Val	Class	Id	Node	Flag

Figure 8. New data structure of attribute list.

Figure 10. PC cluster.

with two attributes. A tree after the division of the root node into node 1 and node 2 is shown in fig.9(b). The record numbers are attached to the nodes to which they belong. The attribute list at the time of Fig.9(b) is shown in Fig.9(c). Here, the class and flag field of an attribute list are omitted. Two attribute lists Attr#1 and Attr#2 have eight records, and they are distributed to 4 PEs. Assume that ($Attr\#1 < 14$) for node 1 and ($Attr\#2 < 3.3$) for node 2 are selected as the dividing condition. According to these conditions, partial translation tables are calculated in each PE in Fig.9(d). The partial translation table of PE 0 shows that the records 3 and 4 belong to node 3. The partial translation table of PE 1 shows that the records 1 and 7 belong to node 4, and the record 5 and 8 belong to node 5. By merging the partial translation table generated by each PE, the whole translation table is generated as shown in Fig.9(e). Merging is implemented by an all-to-all communication function. The decision tree and attribute lists which are updated using the translation table are shown in Fig.9(f). This update can be performed by each PE.

5. PC cluster for data-mining

We selected a PC cluster as a platform for parallel processing because of the cost/performance and scalability. Our PC cluster consisted of 16 PE nodes. The specification of the node PC was determined in consideration of handling of large-scale data. Each node had two CPUs, one of which was used in this experiment. The memory was 256 MB. Since the motherboard, Tyan S1837UANG, with a 440GX chip set, was chosen, the main memory could be extended to 2 GB per node. Each PC was connected by a Switch-

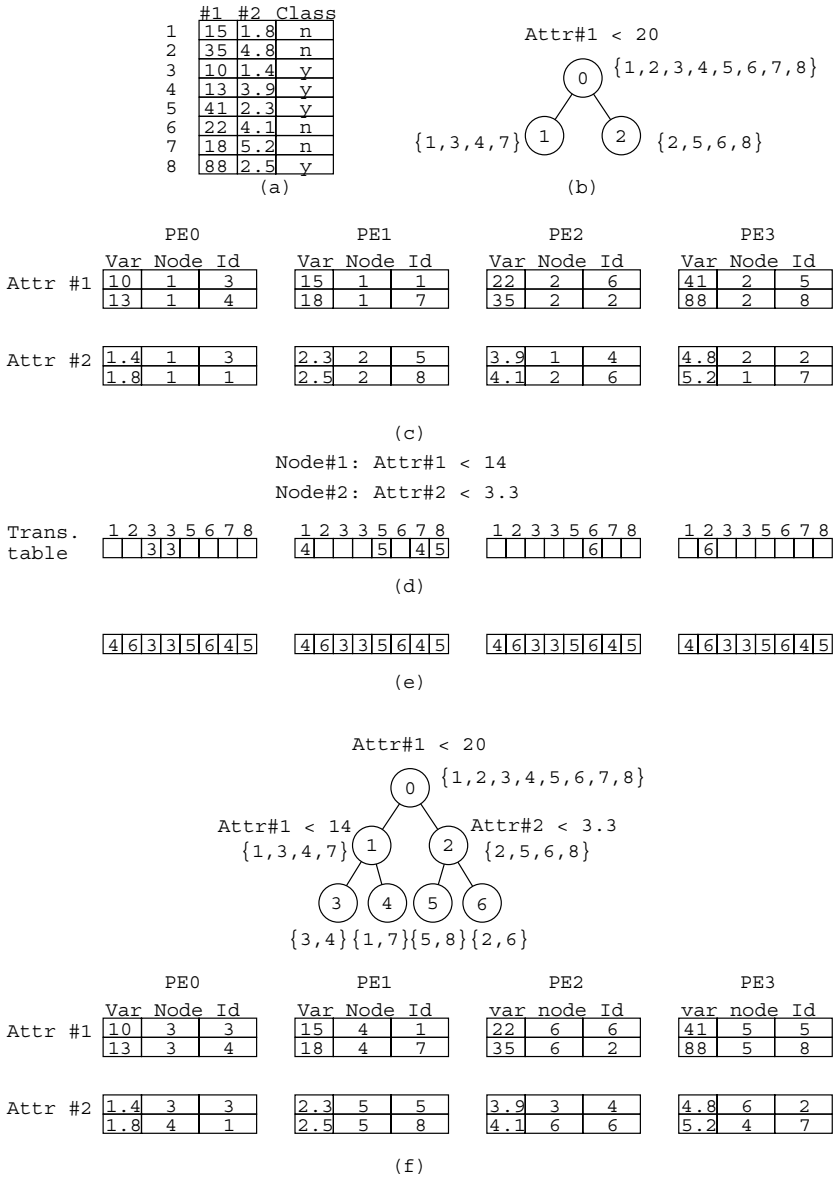

Figure 9. Node generation and translation table update.

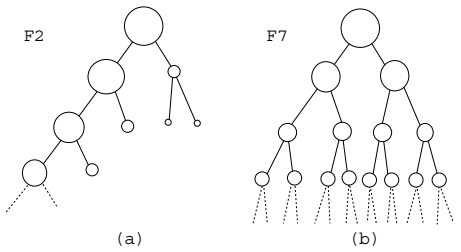

Figure 11. Shape of generated trees. (a) F2. (b) F7.

ing HUB (Catalyst3500) which had 24 ports. LINUX RedHat6.1 (2.2.12kernel) was used as the OS. A photograph is shown in Fig.10.

6. Performance evaluation

Basic performance and the effect of dynamic scheduling of our parallel decision tree was investigated. The effect of the communication performance of a PC cluster to the total processing time of a parallel decision tree is discussed.

6.1. Benchmark data

F2 and f7 of the synthetic database as shown in reference [1], were used as benchmark data. 10 M records with a file size of about 400 MB were used in order to evaluate the effect of dynamic scheduling. The effect of dynamic scheduling tends to be large when the generated nodes can be processed in memory at an early stage. Data for 10 M records cannot be processed in memory with 8 PEs; however, they can be processed in memory with 16 PEs. Therefore, in the case of 8 PEs, it was expected that the node could be processed in memory, in the early stages.

An unbalanced tree was generated from f2, and a balanced tree was generated from f7 (Fig.11). The size of the circle shows the data size in each node. The height of the tree generated from f2 was 26, and the number of nodes was 207. The height of the tree generated from f7 was 36, and the number of nodes was 25267. Since the large data size node remains until the late stages in f2, the effect of dynamic scheduling cannot be expected. On the other hand, the node is divided evenly in f7. Hence the data size of a node becomes small in the early stages, and the effect of dynamic scheduling can be expected.

6.2. Experimental conditions

MPI (mpich1.2.0) was used as the communication library. The input data was placed on the local disk of each PE at first, and then preprocessed. Although the preprocessing execution time is not little, we omit the preprocessing discussion and focus on the tree generation phase. The number of PEs was changed with 1, 2, 4, 8, and 16. The usable total memory size in each case was 256 MB, 512 MB, 1024 MB, and 2048 MB, respectively. The redistribution coefficient was set to 1.2.

Two programs are used for the experiment. One is the parallel decision tree program without dynamic scheduling, and the other is the program with dynamic scheduling. We call them the non-scheduled version and scheduled version, respectively.

Table 1. Execution time of f2 (sec)

PEs	Tree	Calc.	Table	Rebal.
1	10966	10966	0	0
	10750	10750	0	0
2	5520	5406	55	59
	6139	5952	59	128
4	2593	2443	109	41
	2802	2589	113	100
8	1502	1307	162	33
	1413	1206	144	63
16	535	306	208	21
	529	306	206	17

Upper: Non-scheduled version
Lower: Scheduled version

Table 2. Execution time of f7 (sec)

PEs	Tree	Calc.	Table	Rebal.
1	37073	37073	0	0
	28525	28525	0	0
2	18397	18171	196	30
	14774	14517	186	71
4	9662	9196	375	91
	3398	2935	356	107
8	4846	4229	552	65
	1839	1210	550	79
16	1295	536	717	42
	1291	536	716	39

Upper: Non-scheduled version
Lower: Scheduled version

6.3. Parallel efficiencies

The processing time of two programs by changing the number of PEs is shown in Tables 1 and 2. Tree generation consists of computation, table generation, and redistribution. Tree generation time is influenced by the shape

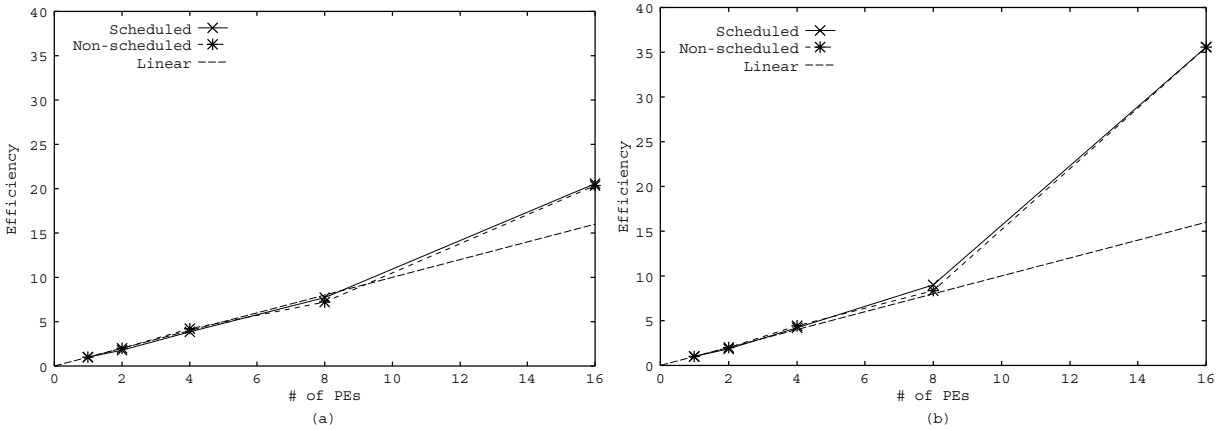

Figure 12. Efficiency for f2. (a) Tree generation. (b) Computation.

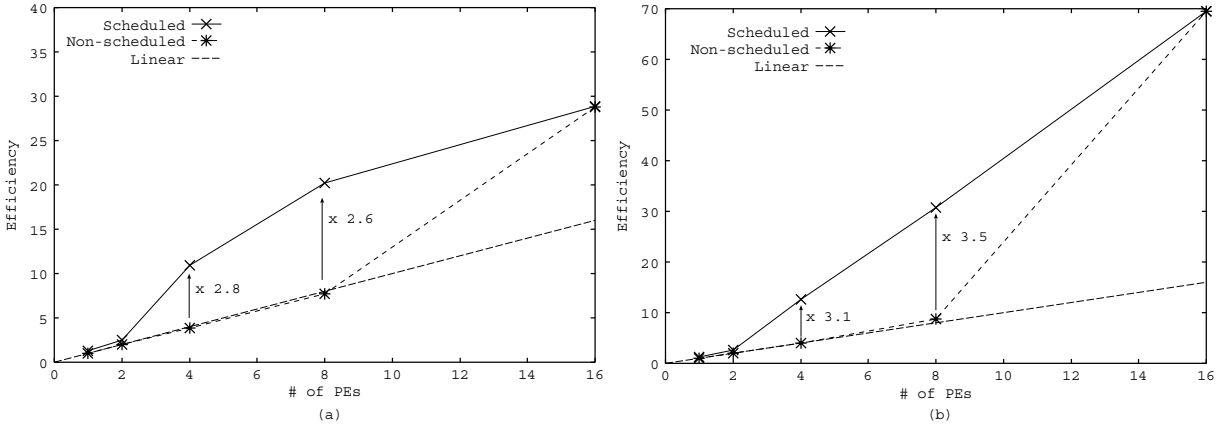

Figure 13. Efficiency for f7. (a) Tree generation. (b) Computation.

and the depth of the tree. The tree generation time of both programs for f2 took about 11,000 seconds, and that for f7 took about 28,000 - 37,000 seconds with 1 PE. The parallel efficiency of tree generation and computation of f2 and f7 are shown in Fig.12-13. The efficiency represents the ratio of the execution time to the execution of 1 PE of the non-scheduled version. F2 and f7 can get super-linear performance enhancement in both programs, with 16 PEs. This is because the data can be processed in memory from a root node execution. In f2, the tree generation became about 20 times faster, and the computation became about 35 times faster. In f7, the tree generation became about 29 times faster, and the computation became about 70 times faster. In f2, both programs showed almost equivalent performance in any number of PEs. This is because a node with large data size remained in the deep level, and the size of the other nodes at the same depth was small enough. In f7, compared with the non-scheduled version, the performance of the scheduled version was enhanced with 4 PEs and 8 PEs. This is the effect of the dynamic scheduling. Although each node can be processed in memory in the early stages of tree generation with 4 PEs and 8 PEs in f7, it cannot be processed in memory by the non-scheduled version because of the grouping. The difference in computation performance between the non-scheduled version and scheduled version was 3.1 to 3.5 times, and that of tree generation was about 2.6 to 2.8 times.

6.4. Influence of communication performance

The breakdown of the tree generation time of f2 and f7 is shown in Fig.14. It can be shown that the time for generating a translation table increases as the number of PEs increases. For both programs with 16 PEs, the table-generation processing time occupied about 20% in f2, and occupied about 60% in f7. It is expected that the table-

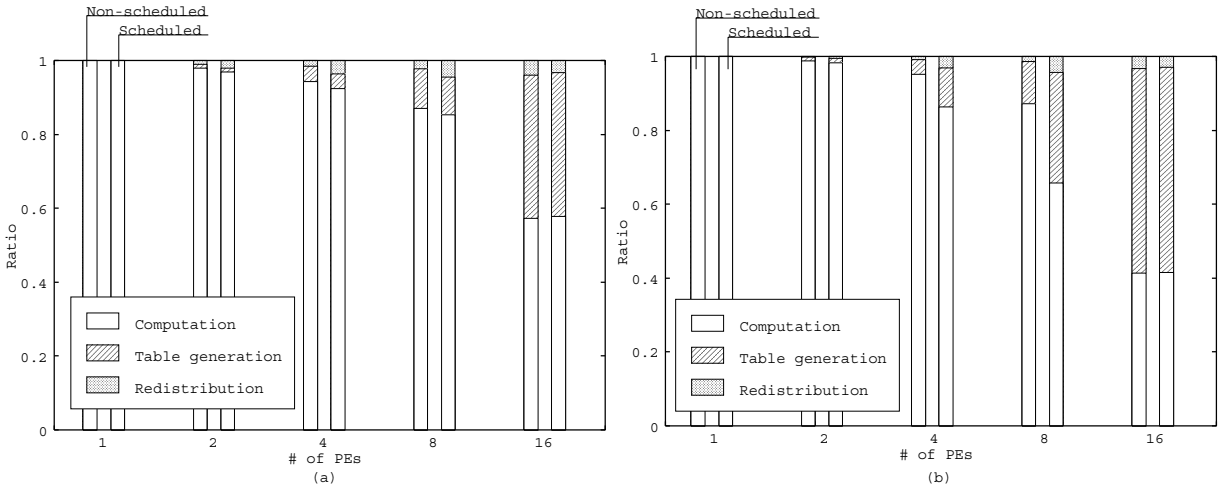

Figure 14. Breakdown of tree generation time. (a) F2. (b) F7.

generation processing time would increase further as the number of PEs increases. Since the communication between PEs takes the major amount of time for translation table generation and redistribution, performance improvement in communication facility on the PC cluster can contribute to the speed increase of the tree-generation time.

7. Discussion

In the translation table generation, reduction of big arrays is required. It can be accelerated by extending the communication bandwidth of a platform. Currently, our PC cluster uses the 100 Base-TX network. By replacing this with a gigabit ether network or myrinet[3], it is possible that the processing time for table generation could be accelerated by more than 10 times. Another speed improvement point is the processing overlap. An overlap of file I/O and computation, and an overlap of communication and computation are possible. By using these techniques, this parallel decision tree generation might be further accelerated.

A dual CPU machine was used for the node of the PC cluster in consideration of the cost performance. It is possible to assign a process for each CPU for parallelization. But, since our program may consume too much memory, parallel efficiency may decrease. Instead, thread programming seems to be more effective.

In the implementation of our parallel decision tree generation, a whole translation table is copied to all PEs. Scalability to the amount of data is lost by this method. If a translation table is divided for every PE, the amount of table to be held by each PE is decreased to 1/PE. However, extra processing is necessary. There is an approach whereby a PE accesses the distributed translation table by remote access. By this method, scalability to the data size is achieved. If there is a high-speed communication facility for remote access, practical performance is possible by the distributed translation table method.

8. Related works

There are various sequential decision tree algorithms: Cart [2] and ID3 [8]. ID3 is the predecessor of C4.5. Cart and C4.5 were improved by NASA to IND-Cart and IND-C4, respectively [6]. Darwin system is the parallelized version of Cart. Mehta et al. developed SLIQ [5] and Shafer et al. developed SPRINT [10]. SLIQ and SPRINT are parallelized for distributed memory parallel computers. Joshi el al. improved SPRINT to ScalParC, which guarantees its algorithmic scalability. SPRINT was also parallelized for SMP [12]. In Srivastava's work [11], efficient parallelization based on formulation of execution and communication speed is discussed. In the survey paper [7] by Provost et al., works for scalability in inductive algorithms is summarized.

We developed and evaluated a parallel decision tree system, which is a parallelized version of C4.5 for SMPs([4]). In this paper, a parallel decision tree generation algorithm was implemented for a PC cluster, and its performance was evaluated. A dynamic scheduling technique with consideration of the memory size was proposed, and its effect was investigated.

9. Conclusion

In this paper, a decision tree algorithm was parallelized for a PC cluster. We implemented a parallel decision tree

generation algorithm based on a SPRINT algorithm, incorporated with a dynamic scheduling function and data redistribution.

The effect of the dynamic scheduling was influenced by the record number and the tree shape generated. From the experiments, the performance was almost the same as SPRINT algorithm in the unbalanced tree, and a maximum speed increase of 3.5 times was achieved in the balanced tree. It was clear that improvement of the communication performance of the platform was important for the speed increase in tree generation for the large number of PEs.

We are planning to evaluate our algorithm by studying the influence of parameters including number of PEs, memory size, data size, and algorithmic variables such as the redistribution coefficient. We are also planning to improve the algorithm on a platform with high communication performance, and develop a system which can treat data from several hundred Giga bytes to several Tera bytes, in practical time.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. *IEEE Transaction on Knowledge and Data Engineering*, 5(6):914–925, December 1993.

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

[3] http://www.myri.com.

[4] K. Kubota, A. Nakase, H. Sakai, and S. Oyanagi. Parallelization of decision tree algorithm and its performance evaluation. In *HPC-ASIA 2000*, pages 574–579, 2000.

[5] M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A fast scalable classifier for data mining. In *the 5th International Conference on Extending Database Technology*, 1996.

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

[7] F. Provost and V. Kolluri. A survey of methods for scaling up inductive algorithms. *Data Mining and Knowledge Discovery*, 3(2):131–169, 1999.

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

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

[10] J. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *the 22nd VLDB Conference*, 1996.

[11] A. Srivastava, E. Han, V. Kumar, and V. Singh. Parallel formulations of decision-tree classification algorithms. *Data Mining and Knowledge Discovery*, 3(3):237–261, 1999.

[12] M. J. Zaki, C. Ho, and R. Agrawal. Scalable parallel classification for data mining on shared-memory multiprocessors. In *IEEE International Conference on Data Engineering*, 1999.

Towards Network-Aware Data Mining

Srinivasan Parthasarathy
Computer and Information Science
Ohio State University,
Columbus, OH 43235
srini@cis.ohio-state.edu

Distributed data mining algorithms executing on a shared network of workstations often suffer from unpredictable performance problems due to limited network resources that are being shared. We show that data mining algorithms, which have an approximate nature, can adapt to network-resource constraints. We argue that existing network monitoring and quality of service mechanisms are insufficient to support the needs of such applications. We then discuss the mechanisms (systems support) needed to support such network-aware applications. Finally, we describe the current status (work-in-progress) of the system under development and discuss some general issues involved in developing resource-aware data mining algorithms.

Keywords: Flexible QoS, Resource-Monitoring, Distributed Data Mining, Shared Cluster Computing

1 Introduction

As our ability to collect, store, and distribute huge amounts of data increases with advancing technology, discovering the knowledge hidden in these ever-growing databases has become a pressing problem. This problem referred to as data-mining, an effort to derive interesting conclusions from large bodies of data, is an interactive process. In fact, interactivity is often the key to facilitating effective data understanding and knowledge discovery. In such an environment response time is crucial because lengthy time delay between responses of two consecutive user requests can disturb the flow of human perception and formation of insight. However, extracting knowledge from these massive databases is a compute and data intensive process which makes the task of guaranteeing quick response times difficult. In order to solve this problem researchers have taken a two pronged approach. To minimize the I/O traffic involved in these applications researchers have evaluated the viability of using data reduction techniques such as discretization, wavelet transforms, and sampling, while sacrificing little in terms of result quality. Simultaneously to compute results faster, researchers are turning to effective parallelization of existing data mining algorithms [19, 23, 3].

Modern-day enterprises usually contain a cluster of shared memory workstations connected by some (intra-enterprise) network. Such a cluster of shared-memory symmetric multi-processors (SMPs) can be a cost effective powerful computational resource. However, the performance achieved by parallel programs on a non-dedicated network of workstations is unpredictable and often leaves a lot to be desired. This is because the performance is affected by dynamic contention for processors, network links, and I/O resources. As an example, for programs using the transmission control protocol/internet protocol (TCP/IP for short), a small amount of contention can play havoc with performance, due to TCP/IP's inbuilt contention-avoidance mechanisms that can reduce communication rates drastically resulting in many idling processors.

One approach to deal with such resource contention is to police the allocation of such resources and having applications adapt to resource constraints. Policing resources requires appropriate mechanisms [12, 21], to arbitrate, allocate, and enforce resource reservations while providing feedback to applications regarding available resources, so that they may adapt. In this paper we focus on how this approach can be adopted for data mining applications when *network resources* are at a premium.

The availability of network monitoring services [5], and quality-of-service (QOS) aware network protocols [6, 4] to police network resources, suggests a natural solution. The problem with directly using existing QOS mechanisms is that they are based on some assumptions that do not hold in the data mining context. These mechanisms have largely been developed for constant bit-rate, low-bandwidth media flows in unreliable network protocols [13, 7]. Interactive data mining applications often exhibit *bursty* traffic patterns, operate on large *remote* datasets, and are typically implemented on top of *reliable* networking protocols. Therefore existing mechanisms are unlikely to suf-

fice. Also, QOS mechanisms for accessing large remote data stores has not been considered in the literature. For data mining applications which operate on tera-bytes of data, this is extremely important.

In this paper we make the following contributions. We demonstrate that data mining applications that are cognizant of network constraints can take advantage of knowing about these constraints and can adapt accordingly. We then consider what mechanisms are lacking and therefore necessary for application adaptability and improvement in overall application performance (on a non-dedicated network of workstations). This latter aspect is *work-in-progress*.

In the next section (Section 2) we sketch our distributed architecture and describe existing QOS work in the IP context, upon which we are building our network QOS-aware support . Then in Section 3 we present how two representative distributed data mining applications can adapt to fluctuations in network performance, motivating the need for effective systems support. Also in this section we pinpoint some of the unpredictable traffic patterns that argues for providing a more flexible Network QOS-aware interface than is currently supported for media applications. In Section 4 we describe the details of the system under development. In this section we describe an overview of the overall resource-aware system, then describe the intended QOS support for local networks and remote data access, including a very preliminary interface. In Section 5 we highlight some critical research issues in translating application-level QOS to system-level QOS. Section 6 documents relevant related work pertaining systems support for data mining applications. Finally we conclude in Section 7.

2 Background

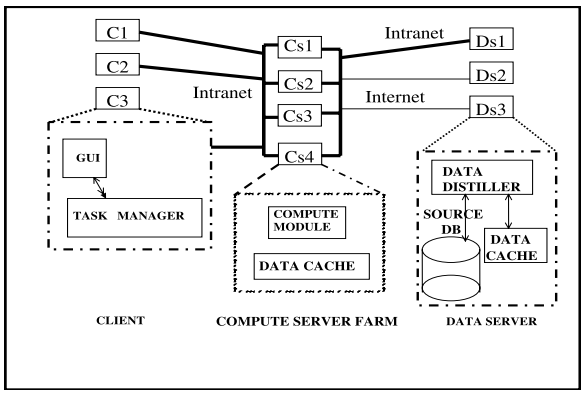

Figure 1. System Architecture

We envision an environment in which a cluster is shared amongst multiple users in a voluntary and cooperative fashion rather than one which is centrally managed and operates on the single program executing at anytime constraint. The latter scenario typically suffers from a gross under-utilization of resources.

In this section we sketch (see Figure 1) the architecture of the distributed data mining system we have in mind. The design of our system took into account the interactivity and large datasets involved in mining applications. The design also takes into account the fact that in many of these applications it is often possible to provide succinct descriptions of the input required (e.g."partition dataset X into 4 clusters" or 'discretize the continuous attributes in dataset Y"). This enabled us to decouple task description from the actual data required by the task and is reflected in our decoupled architecture. This decoupling is important as it potentially enables different tasks to obtain data from multiple sources simultaneously.

Our architecture consists of the following logical components:

- **Client**s (C#) consisting of the GUI, a task manager that directs the mining process, and a local cache (not shown) of data/results of prior computations. It is responsible for the interacting with the data mining engine in terms of invoking, guiding and monitoring computations as well as visualization of the results.

- **Compute Server**s (Cs#), each consisting of a task manager (not shown), a compute module, which is the core data mining engine, and a local data cache. The compute servers and the clients are typically interconnected with each other on a high bandwidth low latency interconnect (Intranet).

- **Data Server** (Ds#) consisting of a data distiller and the source database. The data distiller reads data from the database and performs appropriate data compaction transformations if required (more on this in Section 4.3). We envision that these data servers may be accessible via the local Intranet or via the commodity Internet (low bandwidth, high latency). Although the data servers in the figure are represented by a single box, we envisage that the data servers are based on a cluster architecture, and we take advantage of this assumption, when we describe QOS support for remote data access (Section 4.3).

The physical layout of these logical components depends on available resources. In a fully distributed mode the components, i.e. the client, each compute server and the data server are physically separated. When the data sets are relatively small and the client is relatively powerful, all components could be resident on the client. A hybrid model may combine the compute and data server operations keeping the clients distinct.

As mentioned earlier it is our intention to provide a set of mechanisms that can enable data mining applications to dynamically adapt to network (Intranet and Internet) resource constraints on such an architecture. Since most distributed mining applications are implemented on top of TCP/IP we next discuss existing mechanisms within the context of TCP/IP.

2.1 QoS in IP

We briefly review the current state-of-art in network QOS mechanisms within the context of IP. In IP, packets typically make their way from source to destination via a series of routers. Each individual router must decide which, of usually several packets, to forward as a function of the perceived QOS. The two most common approaches to QOS that have been proposed is the Integrated Services approach [6] and the Differentiated Services [4] approach.

The Integrated Services Approach, requires reservations to be placed by end points on a router by router basis. Each router is responsible for and treating for identifying and treating each application-flow separately according to individual QOS requirements. The Differentiation Services approach, limits the burden on routers by having only edge routers classify which packets should receive better services and then place the class label in in the header. For instance, if a better quality service is to be provided to video-on-demand packets, the edge router will mark it accordingly. Then interior network nodes simply need to aggregate packets based on classes of service and ship them out according to their level of importance. This approach improves over the Integrated Services approach by simplifying the task of routers.

In addition to classifying and marking packets, edge routers may perform policing and shaping, to ensure that senders do not send too much high quality traffic too fast, and to handle bursty traffic, respectively. Policing is accomplished normally by a token bucket mechanism. The size or number of tokens in the bucket controls how quickly an application can send data. Policing is very useful for *enforcing* quality of service. Shaping is accomplished by smoothing out bursty traffic and is done to reduce packet loss.

Mapping these parameters (bandwidth, token depth) for media applications that generate a fixed amount of data (say 8Kb) per frame, at a fixed rate (say 15 frames a second) is pretty straightforward (bandwidth 120 Kbps, token depth 8Kb). The communication patterns associated with data mining applications, described in the next section, are significantly more complex, while the requirements often may be less stringent.

3 Application Adaptability

Data mining applications and the human user driving them, often have varying needs in terms of quality and performance. However, applications (and users) can often adapt to resource constraints by trading off result quality for performance (improved response time), when resources are constrained. In other words the result quality can be sacrificed in a controlled manner when resources are at a premium and response time is crucial. Below we give specific examples of how two applications, discretization and dataset clustering implemented on our architecture can adapt to constraints in network resources.

3.1 Discretization

In the process of growing a decision tree [17], the problem is to determine which leaf node to split and for each leaf node, which attribute to use for the decision at that node. Consider the computation at a node. While typically, a single attribute is used as the decision variable, one can well consider extensions to more than one base attribute (e.g., $X > 5 \wedge Y < 6$) as long as the decisions remain simple [15]. Limiting oneself to selecting base attributes pair-wise, the problem is to determine $f(X_i, X_j)$ for all X_i and for all pairs X_i, X_j, where $\{X_i\}$ are the attributes and $f(X_i, X_j)$ measures the goodness of X_i, X_j as a decision attribute. This problem is referred to as 2-Dimensional Discretization and is computed by the following algorithm:

FOR each pair of attributes
  Step 1: Compute the probability density function (pdf) induced by the 2 base attributes.
  Step 2: Compute optimal discretization (based on some goodness function like entropy)
  Step 3: Score (entropy value) and save result
END FOR
Step 4: Sort results and display top scoring attribute-pairs and corresponding discretizations.

Next we consider how to map this algorithm on to our distributed architecture. Step 1 will typically be computed by the corresponding data server, since this step distills the pairwise attribute columns to a much smaller summary (pdf), that can then be economically transmitted to remote compute servers. Steps 2 and 3 are naturally computed by the local compute servers. Once all the results have been scored and saved, the final step (Step 4) can be executed on the client machine and displayed to the user.

3.2 Network Aware Discretization

This application can adapt to a bandwidth limited environment (as may be the case when the data servers are

Grid Size(MB)	Classification Error
0.00155	13.5%
$0.00625KB$	12.8%
0.025	12.47%
0.1	12.22%
0.4	12.19%
1.55	12.15%
6.25	12.12%

Figure 2. Network-Aware Discretization

truly remote over the commodity Internet) in the following way. The pdf can be computed at different levels of granularity[1]. The finer the granularity the greater the accuracy of the results (lower the classification error). However, at finer granularities the amount of data that needs to be transmitted increases proportionally. We evaluated this tradeoff (results in Figure 2) for a pair of attributes from the LL dataset (described in [15]). From the figure we see that the most accuracy at the highest granularity (grid size = 6.25MB). However, the drop-off in accuracy at a significantly smaller granularity (grid size = 0.1MB) is just 0.1%. This demonstrates that this application can adapt to a bandwidth limited environment by trading off quality (classification error) for significant performance gains (reduction in communication). Note that the total amount of data that needs to be transferred from the data server(s) to the compute servers is equal to $N(N+1)/2 \times gridsize$ where N is the total number of attributes in the dataset. Therefore, the total savings in communication is extremely significant, and comes with a minimal sacrifice to quality. Also note that the top-scoring PDF's are transferred a second time from compute servers to the client (Step 4) resulting in further performance benefits (from working with smaller grid sizes).

3.3 Hierarchical Dataset Clustering

In [14], we define the similarity between two datasets (say D_i, and D_j) to be a function of the difference between the set of associations (A_i and A_j respectively) induced by them ($Sim(D_i, D_j) = f(A_i, A_j)$), weighted by the supports of each association. We also show how such measures of similarity can be used effectively for for clustering homogeneous datasets. The basic structure of this clustering algorithm is shown below:

Step 1 is typically computed by the corresponding data server, since this step distills the datasets to a much smaller summary (association sets A_i). The summaries are then communicated to the compute servers. Steps 2 and 3 are naturally computed by the local compute servers. Once the dataset clusters have been computed, Step 4 can be executed on the client machine and displayed to the user.

[1]The granularity of the pdf estimate is a function of the number of discrete locations at which the pdf is estimated, i.e., the finer the granularity the more the number of discrete locations.

Step 1: Compute associations sets A_i for each dataset.
FOR each pair of datasets D_i, D_j
  Step 2: Compute $Sim(D_i, D_j)$ from A_i, A_j
END FOR
REPEAT (initially treating each dataset as a cluster)
  Step 3: Merge closest pair of clusters
UNTIL Desired number of clusters
Step 4: Display data clusters and similarity matrix

3.3.1 Network Aware Dataset Clustering

TIE (MB)	Accuracy
36	100%
18	99.3%
9	99.0%
7.2	98.9%
5.4	98.4%
3.6	97.8%
1.8	91.5%

Figure 3. Network-Aware Dataset Clustering

In step 2 it is possible to sample the association sets, to limit the amount of information that is being communicated at a cost to accuracy. Table 3 shows the impact of using sampling (which affects the total information exchanged (TIE)) to compute the similarities amongst 12 datasets [14]. Clearly, when the total information exchanged (TIE) is more, the similarity metric is more accurate. However, the loss in accuracy when a tenth of the data is exchanged (3.6MB) is about 2%, which may be tolerated by the user if it does not affect the actual clustering of the datasets (which happens to be the case for this experiment). The merging procedure (step 3) also involves exchanging of complete or partial information (in the spirit of step 2) to compute (or estimate) the association sets for the merged clusters. Similar results, from trading off quality for communication efficiency, was observed for this step as well.

3.4 Other Applications

Classical parallel implementations of various data mining tasks where the data is either centralized or distributed and there is no notion of a separable data server can also adapt to system resources. For association mining [1] researchers have described I/O sensitive techniques that:

trade off disk traffic [18] or communication [23] for extra computation (it is important to realize that quality need not always be sacrificed); trade off disk traffic for quality [22]. Since sequence mining is essentially association mining over temporal data, the same trade-offs can be made for it as well. The same is true for clustering algorithms as well. Recently, we described methods [16] by which the *Expectation Maximization* algorithm can adapt to various resource constraints. We described how this algorithm: can adapt to bandwidth limited environments via lossy and lossless compression; can adapt to available cache and memory sizes via program and data transformations, and can adapt to computational resource constraints via appropriate program transformations.

3.5 Traffic Patterns

In this section we isolate the traffic patterns of distributed data mining algorithms in general while paying particular attention to the two applications we have described. More importantly we identify how these applications differ from typical media applications. As pointed out earlier media applications usually have an easy to specify communication pattern. The communication patterns of data mining applications differ in the following respects:

- **Unpredictable Message Sizes:** The amount of information communicated depends largely on the data characteristics, and input parameters (or user interaction). In some cases the exchange of information is minimal and in yet others it is a large amount. For instance, between Step 1 and 2 of the hierarchical dataset clustering algorithm, the association sets are communicated from the remote data server(s) to the compute cluster. The number of actual associations in a given association set is completely dependent on the characteristics of the dataset [2], as well as on the input parameters (minimum support), and is therefore difficult to predict in advance. Distributed versions [3] of the Apriori [1] algorithm that exchange candidate itemsets also exhibit this property. Here, the number of candidate itemsets, and therefore the amount of information communicated, depends on the characteristics of the dataset and input parameters.

- **Bursty Communication:** As we see from the algorithm descriptions above, data mining applications may compute for a bit, then communicate information, and then compute again. In many applications (such as discretization; when multiple pdf's are mapped to the same compute server) communication (receiving pdf's from data servers) can be overlapped with computation (computing the optimal discretization on the pdfs that have already been received). In other cases computation ceases until communication completes. This results in bursty communication patterns that can overwhelm network resources, especially in the case of data intensive applications like data mining, resulting in poor performance.

- **Varied Communication Models:** In media applications the normal communication model is a simple synchronous, and continuous producer-consumer or one-on-one streaming communication pattern. Data mining applications exhibit a range of communication patterns such as synchronous, asynchronous, pairwise (one-on-one) and one-many (broadcast) and many-one (slave-master).

- **Dependency on Reliable Communication:** Data mining applications by and far rely on reliable network protocols unlike media applications. Communication in such applications is typically achieved via reliable protocols such as TCP/IP. TCP's flow control and congestion control mechanisms while critical to the effectiveness of TCP in shared networks have the unfortunate consequences of making TCP traffic sensitive to the loss of individual packets.

The above issues notwithstanding data mining applications are typically more *flexible* than media applications and can often place slightly less stringent QOS requirements on the network resources. For example in video-on-demand applications if a certain frame rate cannot be guaranteed, then the quality fallout may be unacceptable. On the other hand in data mining applications, there is often, and specifically in the case of the two examples we outlined above, no strict bandwidth requirements (as in requiring X bits per second). Therefore, the system can be more flexible in terms of bandwidth guarantees, and admission control. As a result even when the traffic patterns are bursty, mining applications are conducive to fairly aggressive shaping policies. The caveat here is that some mining applications, especially active mining applications, that operate on time-varying data might require strict bandwidth, and anti-shaping guarantees. Therefore, data mining applications can benefit *from a system that can support* **both** *conventional (strict) and more flexible classes of QoS.*

4 System Details

In this section we consider the details of the system under development to support network-aware data mining. We first briefly overview how we envision our system will process and control resource reservations. We then discuss the flexible QOS classes that we plan to support, for data

[2] It depends on the average number of items per transaction, total number of items in the dataset, and actual associativity of the dataset.

mining applications, within the local compute cluster. We then describe how we support QOS for remote data access.

4.1 Overview of the System

Static information available to the system includes the type of connection within the compute cluster (peak bandwidth, minimum latency),and similar information for connections to the the data servers. Dynamic information available to the system include current resource utilization, and future resource utilization (from application reservations). At any given point the network *resource scheduler* can receive a resource reservation request from a client. The scheduler can also simultaneously receive similar information from other clients. Based on all the new requests, existing client priorities (briefly described in the next section) and the available resources at the cluster, the scheduler needs to decide on the optimal set of services to be scheduled. Depending on the optimization function (some combination of maximizing throughput of the system, best response time for prioritized clients, and guaranteeing QOS) the scheduler needs to derive the final set of resources (and corresponding QOS) that can be reserved. Once the scheduling decision is made, the admitted client applications are notified of the resources that are available and the corresponding QOS available. The scheduler also informs the *resource enforcer* about the admitted jobs which in return enforces the respective QOS for the clients. Based on this information, especially if the reserved resource is not at the desired QOS requested by the client, the application can adapt to the available service from the cluster. Once the reservation related to a particular resource is no longer required, the resource is released back to the cluster. We next describe the QOS interface for the compute cluster and the QOS interface for remote data access.

4.2 QOS for Local Networks

The QOS classes that we intend to support for data mining applications are *short low latency, premium (2), best-effort and 0-priority*. *Short low latency* messages are for short high priority messages that can be used for sending signals or for collective synchronization operations like barriers. *Premium messages* can be used for high priority messages that are too large to be supported by the low latency class of service. Based on the discussion presented in Section 3.5 we plan to support two forms of premium messages, i.e., *with and without bandwidth guarantees*. Data mining applications, requiring a strict QOS requirement, can use the former class (with bandwidth guarantees). Other applications, where the bandwidth requirements are not as stringent (or unknown), can use the more flexible premium message class (without bandwidth guarantee) while still getting a reasonable level of QOS. Under the latter class, the system also has the flexibility (within moderation) to police and shape according to dynamic network information. *Best effort messages* can be used to indicate which messages do not require QOS service and can therefore be aggressively shaped or policed by the system. *0-priority* messages are messages that can be dropped when contention in the system becomes an issue. Such messages can be used in data mining in applications where the arrival or non-arrival of a message does not affect the produced results. For instance in association rule mining, pruning of candidate itemsets can improve the search speed [1], however, if there is heavy contention in the network then the information required by the pruning algorithm may be delayed, thus delaying the overall algorithm. If instead one were to send all pruning information using a one-way asynchronous 0-priority message, then if there is heavy contention this information would not get through else it will. The algorithm can tolerate this message loss, with no loss to result quality at the cost of extra computation.

There are other parameters involved when placing a network resource reservation. The *bandwidth* parameter is required, for one class of premium messages, and it reflects the amount of information that needs to be sent per unit time. The *total message size* (if known or if it can be estimated[3]), can allow the system to compute the the projected reservation time based on current system state. This is useful for admitting other reservations (much like how restaurant reservation systems work).

There is also a need to provide mechanisms which would allow applications to monitor reservation requests. If the requested class of service is unavailable the system will return the next best available class of service that it can currently guarantee. The application can query the reservation, and if the desired QOS level is not available, then it can decide how to adapt. A sample piece of code below describes how a programmer could take advantage of the current interface.

```
QOS_OBJECT *QO = QoS_adv_reservation(PREMIUM1, bw, message_sz,...)
/* ... other information such as source, destination etc. */
.....
QOS_RESERVATION_DATA *QOR = QO->get_reserved_info()
.....
IF (!QOR->got_requested_reservation())
     Read what we got
     Adapt accordingly
     QO->Send(*msg_ptr);
ELSE
     Continue as per plan
     QO->Send(*msg_ptr);
ENDIF
.....
QO->relinquish(); /* required for PREMIUM1 service, implicit for others */
```

[3] We have pointed out in Section 3.5 that the total message size is difficult to guess apriori in many data mining applications. However, in some cases this information can be estimated, either via sampling or from historical performance data, slightly in advance of when the message is actually sent.

The code first places an asynchronous advance QOS request (in the spirit of asynchronous I/O requests) with the desired class of service ($PREMIUM1$), the desired bandwidth (bw) and the estimated size of the message ($message\_sz$). Placing advance reservations is recommended especially if requesting premium or low-latency class of service as the system has more time to make a decision. Such requests will return a communicator object handle (QOS_OBJECT), along with the exact QOS that will be guaranteed. The application can view the guaranteed reservation (using the $get\_reserved\_info()$ function). This information can be used by the application if it needs to adapt to a lower than expected guarantee of service. The $Send()$ function is used to send the corresponding message(s). Relinquishing reservations are implicitly handled by the system once the message has been sent, for all classes of service except premium messages with bandwidth requirements. For this class of service the reservation will have to be relinquished explicitly (as shown in the code fragment using the $relinquish()$ function).

4.3 QOS for Remote Data Access

QOS over the commodity Internet is difficult to guarantee. Perhaps the most challenging aspect of a remote storage server design lies in the bandwidth barrier represented by the wide area link protocols. Theoretically the bandwidth on the (wide-area) Gigabit Ethernet is comparable to that of a local (campus-wide) cluster. However, the well documented overhead of conventional TCP/IP protocol stacks limits the link data rate to values around 50 MB/s, no matter how fast the wire.

To overcome this problem, we have adopted a parallel communication approach in the spirit of [10] and [2]. The basic idea is to communicate by striping data over parallel TCP/IP connections. Our scheme improves on the above in that each of the connections can use distinct endpoint nodes since we envision that each individual data server is basically a cluster architecture. Basically, by taking advantage of the presence of clusters on both ends of the geographical link, we get around the bottleneck by effectively parallelizing the TCP/IP protocol processing. The scheme scales up with the number of nodes used for each cluster, up to the physical limit of the intervening link. An additional software layer is in charge of striping and de-striping on the two ends, while taking advantage of the fast user-level communication available on the local clusters.

The above approach also enables applications to specify a quality of service in terms of the number of striped connections. The quality of service can be controlled by increasing or decreasing the number of striped connections in response to changes in application requirements. The two classes of service that we support here are *best effort* and *premium* (with striping parameters). When requesting the premium class the application can specify the striping parameters that will correspond to the number of striped connections per node as well as the number of nodes involved for each cluster. The network scheduler will determine whether the request can be met (which will involve communicating with its corresponding unit at the data server end).

Table 1 presents some preliminary experiments in which we evaluate the the impact of the above approach. The first column (#CS), represents the number of compute server nodes involved in the experiment. The second column (#RS) represents the number of remote data server nodes involved in the experiment. The third column (#PC), represents the number of parallel communication channels between the remote server(s) and compute server(s). The second row of the table represents an experiment involving four parallel connections between one compute server node and one remote server node. Finally, the fourth column represents the total time for the data transfer. This includes the time for communication, de-striping, and storing on the local file system. We assume that the data is already striped on the remote servers.

#CS	#RS	#PC	Total Time (s)
1	1	1	26.4
1	1	4	24.1
1	2	4	22.6
1	4	4	21.1
2	1	4	22.1
2	2	4	21.3
2	4	4	20.1
4	1	4	21
4	2	4	20.3
4	4	4	18.6

Table 1. Evaluating Striping for Remote Data Access

Our experimental set-up consisted of compute servers located in the Ohio-State University. The data servers were located at Rochester, NY. The communication was over the commodity Internet. In all the experiments 16MB of data was transmitted. The overall performance was severely inhibited by the fact that compute cluster has to make connections to the outside world via a single gate-way node. In spite of this limitation we observed encouraging results. On going from 1 (row 1 (*best-effort*)) to 4 (row 2) parallel connections for one compute node and one data node we see that there is a 10% improvement in performance. As we pointed out earlier, increasing both the number of remote server nodes (rows 2-4), and the number of compute nodes (rows 5-11) resulted in further improvements in performance (10-35%).

Another form of QOS that we plan for remote data access is *data-preprocessing time* at the data-server end. As we pointed in Section 3, many data mining applications can use data reduction operations like wavelet transforms, pdf generation, various time/frequency transforms, sampling, etc., to reduce the amount of data that needs to be transferred thereby ensuring that even with lower bandwidth reservations the reduced data can be obtained in a timely fashion. However, these data reduction operations can be computationally intensive. In order for the data reduction operations to be effective the sum of the time taken to compute the reduction and the time taken to send the reduced data must be less than the time to send the original data. Therefore, the application needs to place an appropriate QOS reservation for computational resources [12, 21] on the data server.

4.4 Current Implementation Status and Issues

We are clearly at the very early stages of implementing the ideas presented in the previous few sections. We are in the process of implementing the interface for local area networks based on the Differentiated Services-based system, where the QOS class labels (premium, short-low latency etc.) are placed in packet headers, so that policing can be enforced and shaping of messages can be carried out where appropriate. After the basic interface has been implemented we will implement the meat of the system, viz., the resource scheduler. For remote data access, we have implemented the quasi-QOS protocol as outlined above and are evaluating it.

One interesting research issue we are investigating is to identify a mechanism that can handle premium messages with bandwidth guarantees effectively. One possible solution we are investigating is to estimate and dynamically adjust the optimal depth of the token bucket at the edge routers. We are evaluating the viability of estimating this information using current premium bandwidth reservation information coupled with dynamic network performance data.

5 Translation of user level QoS to System-level QOS

To provide a particular level of user/application level QoS, there is a need to translate the user QoS to the underlying system-level QoS presented in the previous section. This translation will naturally have to be done in an application-specific manner. Below we discuss the research issues involved in this translation process.

For data mining applications, the user's requirements are mostly specified in terms of performance and quality of results (Table 2 and Table 3). Performance is usually described as the amount of time that is allowed to fulfill the user's request, i.e., *response time*. Based on the algorithm that is involved, this response time specification can be translated into the deadline for completing the requested operation. This "execution deadline" information can be used by the scheduler to *prioritize* among competing clients requesting network resource reservations. To specify the quality requirement, the user can typically specify the amount of error (E) that is tolerated. The relationship (network resources = f(E)) between the error tolerance and the required network resources is application-specific. This relationship can often via static or run-time profiling on sampled data be estimated. For instance on knowing the maximum tolerated classification error for building a decision tree one can, after sampling the data at runtime, estimate the amount of data that needs to be communicated in order to guarantee this error tolerance requirement. By knowing the amount of data that needs to be communicated, and in conjunction with response time requirements of the client, one can compute the appropriate bandwidth requirements needed, which can then be reserved.

6 Related Work

Several systems have been developed for distributed data mining. The JAM [20](Java Agents for Meta-learning) and the BODHI [11] system assume that the data is distributed. They employ local learning techniques to build models at each distributed site, and then move these models to a centralized location. The models are then combined to build a meta-model whose inputs are the outputs of the various models and whose output is the desired outcome. The Kensington [9] architecture treats the entire distributed data as one logical entity and computes an overall model from this single logical entity. The architecture relies on standard protocols such as JDBC to move the data. The Id-Vis [21] architecture is a general-purpose architecture designed with single data mining applications in mind to work with clusters of SMP workstations. Both this system and the Papyrus system [8] are designed around data servers, compute servers, and clients as is the system presented in this work. The Id-Vis architecture explicitly supports interactivity through the interactive features of the Distributed Doall programming primitive. However, the interactions supported are limited to partial result reporting and bare-bones computational steering. Our work is complementary to the above distributed data mining systems. Their focus is on how to build data mining systems or specific data mining applications when the data and processing capacity is distributed. Our focus is on how to build systems support for resource-aware data mining.

7 Conclusions

Many data mining algorithms have unpredictable performance on shared computational environment, due to resource constraints. In this paper we present preliminary results showing how data mining algorithms can effectively adapt to network resource constraints if proper systems support and a flexible interface to control and enforce network resource acquisition were available. We have argued that existing QOS mechanisms for media applications are not appropriate for data mining algorithms since the communication needs and application properties are very different. We have then outlined a desired (wish-list) set of QOS mechanisms that ought to be supported in order that adaptable data mining algorithms can monitor and acquire resources to get predictable and desired performance and have described how we are attempting to achieve this goal.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *20th VLDB Conf.*, Sept. 1994.

[2] S. Bailey, E. Creel, R. Grossman, S.Gutti, and H. Sivakumar. A high performance implementation of the data space transfer protocol. In *Workshop on Parallel and Distributed KDD Systems*, 1999.

[3] D. Cheung, J. Han, V. Ng, A. Fu, and Y. Fu. A fast distributed algorithm for mining association rules. *4th Intl. Conf. Parallel and Distributed Info. Systems*, Dec. 1996.

[4] B. et al. An architecture for differentiated service. In *IETF Network Working Group, RFC2475*, 1998.

[5] B. L. et al. A resource query interface for network-aware applications. In *HPDC*, 1998.

[6] Z. et al. A new resource reservation protocol. In *IEEE Network*, 1993.

[7] I. Foster and C. Kesselman. The grid: Blueprint for a future computing infrastructure. In *Morgan Kauffman Publishers*, 1999.

[8] R. Grossman, S. Bailey, S. Kasif, D. Mon, A. Ramu, and B. Malhi. Design of papyrus: A system for high performance, distributed data mining over clusters, meta-clusters and super-clusters. In *Proceedings of Workshop on Distributed Data Mining, alongwith KDD98*, Aug 1998.

[9] Y. Guo, S. Rueger, J. Sutiwaraphun, and J. Forbes-Millot. Meta-learning for paralle data mining. In *Proceedings of the Seventh Parallel Computing Workshop*, 1997.

[10] J.D.Touch. Parallel communication. In *INFOCOMM*, 1993.

[11] H. Kargupta, I. Hamzaoglu, and B. Stafford. Sacalable, distributed data mining using an agent based architecture. In *KDD*, Aug 1997.

[12] C. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Morgan Kauffman Publishers*, 1999.

[13] K. Nahrstedt, H. Chu, and S. Narayan. Qos-aware resource manangement for distributed multimedia applications. In *Journal on High-Speed Networking*, 1998.

[14] S. Parthasarathy, and M. Ogihara. Clustering Homogeneous Distributed Datasets. In Fourth Practical Applications of Knowledge Discovery and Data Mining (PKDD), 2000.

[15] S. Parthasarathy, R. Subramonian, and R. Venkata. Generalized discretization for summarization and classification. In *preprint*, Jan. 1998.

[16] S. Parthasarathy, and R. Subramonian. Adaptive EM-Clustering. Technical Report OSU-CISRC-12/00-TR26, Ohio State University, Nov. 2000.

[17] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 5(1):71–100, 1996.

[18] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *21st VLDB Conf.*, 1995.

[19] J. Shafer, R. Agrawal, and M. Mehta. Sprint: A scalable parallel classifier for data mining. In *VLDB*, pages 544–555, 1996.

[20] S. Stolfo, A. Prodromidis, and P. Chan. Jam:java agents for meta-learning over distributed databases. In *KDD*, Aug 1997.

[21] R. Subramonian and S. Parthasarathy. A framework for distributed data mining. In *Proceedings of Workshop on Distributed Data Mining, alongwith KDD98*, Aug 1998.

[22] M. J. Zaki, S. Parthasarathy, W. Li, and M. Ogihara. Evaluation of sampling for data mining of association rules. In *7th Intl. Wkshp. Research Issues in Data Engg*, Apr. 1997.

[23] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New parallel algorithms for fast discovery of association rules. *Data Mining and Knowledge Discovery: An International Journal*, to appear, Dec. 1997.

Incremental Quantitative Rule Derivation by Multidimensional Data Partitioning

Junping Sun
School of Computer and Information Sciences
Nova Southeastern University
Davie, Florida 33314-4416, USA
Email:jps@nova.edu

Abstract

By using cardinality and relevance information about a set of attributes and concept hierarchies, a top-down incremental data partitioning method is proposed for quantitative rule derivation from database in parallelism. Based on sequential incremental approach, we proposed two parallel versions of incremental partitioning algorithms. These two parallel algorithms are multidimensional-based to partition data set into multiple independent subsets for further rule derivation process. The second version of the parallel algorithm improves the first in terms of load balance.

1 Introduction

Rule derivation has been considered as one of the important research subjects in knowledge discovery and data mining in database systems and machine learning. From academic research point of view, knowledge discovery is referred to as non-trivial extraction and identification of implicit, previously unknown, potentially useful, and ultimately understandable patterns from raw data in database systems [6, 7]. From practical or industrial perspective, data mining is a step in the KDD process consisting of applying data analysis and discovery algorithms that, under acceptable computational efficiency limitations, produce a particular enumeration of patterns over the data [7].

With various methods in knowledge discovery and data mining, these potentially useful and previously unknown knowledge can be discovered and derived from large databases and/or data warehouses. The discovered knowledge from the relevant sets of data in databases can be correspondingly used for information management and monitoring, decision support and making, query processing, OLAP (online analytic processing), and many others.

Huge data volumes in gigabytes, terabytes, and petabytes are no longer fictitious numbers but become real challenge to KDD. The size of today's databases requires not only efficient and effective storage and retrieval of the data for online transaction processing (OLTP), but also for online analytic processing (OLAP) and KDD. The intensive process of knowledge discovery process in large databases, data warehouses, and information repositories requires efficient and intelligent methods that are different from those developed in the fields of artificial intelligence and machine learning [6, 7, 13, 14]. It requires and obligates methods to scale up data mining algorithms, which includes development of novel data mining algorithms, or adaptation and integration of existing data mining algorithms, in order to deal with huge data volumes in KDD process.

Besides alternatives to develop new data mining algorithms, adapting existing ones, or integrating, some approaches try to exploit emerging hardware technology such as massively parallel computer machines that are becoming more popular. Hence, many research endeavors have scaled up data mining algorithms by either changing existing sequential algorithms into parallel versions or developing new parallel data mining algorithms [1, 2, 23, 24]. In this paper, we will propose a top-down incremental data partitioning method for parallel rule derivation which derives a set of multilevel quantitative rules from a relational database table. A multidimensional data structure is used to store not only the derived quantitative rule information, but also to facilitate the update of the quantitative rule information.

2 Literature Review

Sequential approaches for rule derivation from earlier research can be found from the literatures in artificial intelligence, machine learning [13, 14, 15, 18], and data mining and knowledge discovery [2, 3, 6, 7, 8, 9, 10, 11, 12, 17, 21].

The primary research on knowledge discovery in database systems has been the focus on deriving rules such as association [2, 3, 8, 12, 16, 17, 21], classification, characterization, discriminant, description [9, 10], etc. Association rule mining finds all rules in the database that satisfying some minimum support and minimum confidence constraint [3]. Classification rule mining is to discover a set of rules in the database to form an accurate classifier [18].

In many cases, it is hard to find a single crisp rule to classify and to characterize the association among entire

set of data due to complex combination of properties in existing data. So quantitative measurements are used to discern the data set and to derive rules from the data in the sense of probability of the truthfulness with respect to the set of derived rules [3, 9].

Typical quantitative description rule derivation approaches in a universal relational data-base table can be found in [9, 10], where a set of concept hierarchies is used to guide the rule derivation process. The set of rules derived by the methods in [9, 10] is complete by augmenting each of derived rules with quantitative vote in order to classify overlapped data tuples from different classes. Various related research work in mining association rules has been given in [2, 3, 12, 17, 21]. The key issue in mining association rules is how to find the large item sets, i.e., all item sets that have the support above a threshold value: the predefined minimum support, and to use the large item sets to discover the set of associate rules. With support s and confidence c values [2, 3], only the association rules whose support values are greater than the threshold values will be finally generated. The rule derivation approaches can be classified into: supervised [9, 10], unsupervised [3], or semi-supervised [21]. For semi-supervised approaches, they utilize hash functions or pre-defined categorization to classify attributes with numerical data type.

The computation of the rule derivation process is intensive due to huge volume of data, and sometimes exhaustive in order to discover the complete set of all useful rules [2, 3]. Because of these, many parallel and scalable algorithms have been proposed for fast rule derivation [1, 2, 23, 24].

3 Background and Previous Research

In order to facilitate further discussion and problem formulation, we would like to give the following definitions.

3.1 Definitions

Definition 3.1 *Let $R(A_1, ..., A_D)$ or $R = \{A_1, ..., A_D\}$ be a universal relational database schema, where A_i is an attribute or a dimension in the schema, and n is the number of attributes or dimensions in schema R for $1 \leq i \leq D$.*

Definition 3.2 *Let $CH = \{ch(A_i) \mid i \leq D\}$ be a set of concept hierarchies or concept hierarchy trees which specifies taxonomy of concepts on top of primitive data in a database.*

Definition 3.3 *Let $ch(A_i)$ or ch_i be a set of $h_{i,j,k}$s, and each $h_{i,j,k}$ is a concept element in the concept hierarchy corresponding to a node in the concept hierarchy tree, where j is the j^{th} level in a concept hierarchy, and k is the k^{th} specialized element (attribute) for a given j at the same level of a concept hierarchy $ch(A_i)$.*

Without loss of generality, it is assumed that all the concept trees are in the form of balanced tree.

Definition 3.4 *Let $card(A_i)$ be the cardinality (the number of distinct values) for attribute A_i denoted as $\mid A_i \mid$. Let $ch_{i,j} = \{h_{i,j,k} \mid 1 \leq k \leq \text{card}(R)\}$ be set of specialized concept elements of at the j^{th} level in the concept hierarchy $ch(A_i)$ for A_i in R, and $\mid ch_{i,j} \mid$ be the cardinality for $ch_{i,j}$.*

Definition 3.5 *Let $height(ch(A_i))$ be the height for the concept hierarchy $ch(A_i)$ and $\mathbf{max}(height(CH))$ be the maximal height for the set of concept hierarchies such that:*

$$\mathbf{max}(height(CH)) = \mathbf{max}\{height(ch(A_i))\}$$

where $1 \leq i \leq D$.

Definition 3.6 *Let $\mathbf{C}$ be a set of dimension cycles such that:*

$$\mathbf{C} = \{C_j \mid 1 \leq j \leq \mathbf{max}(height(CH))\}.$$

Each C_j is an ordered set such that:

$$C_j = \{d_1, d_2, ..., d_i, ...\}$$

where $C_j \subset \mathbf{N}$, $\mid C_j \mid \leq D$, $0 \leq d_i \leq D-1$, $d_i \leq d_{i+1}$, and $d_{i+1} - d_i \geq 1$.

Definition 3.7 *Suppose n is a node. Let $T_{i,j,1}$, $T_{i,j,2}$, ..., $T_{i,j,k}$ be trees with roots $n_1, n_2, ..., n_k$ respectively. A new tree with n being the parent of nodes $n_1, n_2, ..., n_k$ is constructed in a way such that: n is the root of the tree, and $T_{i,j,1}$, $T_{i,j,2}$, ..., $T_{i,j,k}$ are the subtrees of the root. Nodes $n_1, n_2, ..., n_k$ are the children of the node n. i is the dimension or attribute in the schema, j is the level in the concept hierarchy corresponding to the dimension i, and k is the number of partitions or classifications corresponding to i and j. Without loss of generality, the height of the tree is bound by: height($\mathbf{T}$) $\leq \sum \mid C_j \mid$.*

Definition 3.8 *If $T_{i,j,k}$ is an internal node, then $T_{i,j,k}$ is a n-tuple such that:*

$< count, i, j, k, s_{weight}, c_{weight}, \{ptr_1, ptr_2, ...\}, ... >$

where count is used for computing s_{weight} and c_{weight} stored at each level of the tree, and $\{ptr_1, ptr_2, ...\}$ is a set of pointers. If $T_{i,j,k}$ is an internal node, then the set of pointers that link the partitioned subtrees to the next level. If $T_{i,j,k}$ is a leaf node, then the set of pointers that link to buckets where the set of data tuples belong to the partition associated with $T_{i,j,k}$.

3.2 Previous Related Research

Most relevant research in quantitative rule derivation is the concept hierarchy based and attribute-oriented induction method proposed by Han *et al.* [9, 10]. The relational database table used in the rule derivation process can be a subset of the universal database schema. In the bottom-up quantitative rule approaches

in [9, 10], the set of concept hierarchies is used to guide the process of rule derivation, and the generalization process selectively picks the attribute with larger reduction rate, which means larger support s value and/or implies larger confidence c value [2, 3]. The center of their approach is *attribute-removal* and *concept ascension*. The purpose of *concept ascension* [9, 10] is to remove duplicate tuples by substituting the attribute value with the corresponding one at the next higher level of the concept hierarchy. *Attribute-removal* is to drop the attributes that are impossible to generalize and to support further concept ascension. One of heuristics used in [9, 10] is to find the attribute column with the larger reduction ratio. The importance of using the set of concept hierarchies as background knowledge is to guide the generalization process in the correct direction.

4 Sequential Incremental Partitioning

We start with illustration of our sequential incremental rule derivation method in order to introduce and understand the parallel version of the sequential algorithm. Our sequential incremental partitioning approach for quantitative description rule derivation is different from the data-driven method [9, 10] as described in the following:

1. The previous approach in [9, 10] is bottom-up data driven method. Our method, presented in this paper, is a top-down, goal-driven approach. One of the advantages of the top-down approach is the scalability, i.e., as long as the partition is done along a dimension at one level with a given concept hierarchy, the partitioning at the next dimension can be processed parallelly.

2. Similar to the approach given in [9, 10], our top-down method selectively picks an attribute for data partitioning and derivation with a small cardinality value of the attribute. If an attribute has a smaller cardinality, then more tuples in a table share the same attribute value, the attribute value has a higher frequent appearance or count, and the attribute is more representative as a predicate. The smaller the cardinality value for the attribute is, more relevant the attribute is to the set of derived rules. The smaller the cardinality value is, the larger the support s value is such as: ([2, 3])

$$cardinality \propto \frac{1}{s} \quad or \quad s \propto \frac{1}{cardinality} \tag{1}$$

3. Each time an attribute or a dimension is picked for partitioning or classifying, the data tuples from the same category will be grouped together for further partitioning or classifying along next attribute or dimension.

4. The partitioning or classifying process is cycled based on the cardinality values of attributes or dimensions, and the next dimension *next_d* in the cycle is determined by:

$$next\_d(i) = \begin{cases} (i+1) \bmod D & j \leq height(ch(A_i)) \\ next\_dim(i+1) & \text{otherwise} \end{cases} \tag{2}$$

where i is the current attribute or dimension for partitioning, D is the total number of dimensions and attributes involved in the process.

5. Since the partitioning or classifying method is a top-down process, the first cycle of partitioning or classifying will be based on the classifications at the first level of all the concept hierarchies. It will move down to the next level of the concept hierarchies in the next dimensional cycle $i = 1, 2, ..., D$. The next level $L_{i,j}(j)$ of the concept hierarchy for a corresponding dimension or an attribute A_i in the next dimension cycle can be determined by the following:

$$L_{i,j}(j) = \begin{cases} j & if\ (i+1) \bmod D > i \bmod D \\ & \& \ j \leq height(ch(A_i)) \\ j+1 & if\ (i+1) \bmod D < i \bmod D \\ & \& \ j \leq height(ch(A_i)) \\ \infty & \text{otherwise} \end{cases} \tag{3}$$

where $1 \leq j \leq \mathbf{max}(height(CH))$.

Since it may be true that $height(ch(A_i)) \neq height(ch(A_j))$ for $i \neq j$, the partition along dimension A_i will be skipped if $j > height(ch(A_i))$.

4.1 Sequential Partitioning Algorithm

The top-down partitioning approach for quantitative rule derivation is multidimensional based, and it scans the data and classifies the knowledge along each dimension or attribute by dimension cycling and concept hierarchy descending.

1. The algorithm first generates a set of ordered n attributes A_is such that:

$$| A_i | \ \leq \ | A_j |, \quad i \leq j \leq n$$

This implies the classification process will start with the dimension with the least number of specialized attribute values, which has the smallest cardinality value. If $| A_i | = | A_{i+1} |$, then the order will be determined by the value of $| ch_{i,j} |$ such that:

$$| ch_{i,j} | \ < \ | ch_{i+1,j} |$$

2. As soon as the sequence of attributes is determined, the classification or partition can be performed based on the watershed attributes corresponding to the elements at the highest level in the concept hierarchy.

3. As soon as the partition or classification is done along the attribute or dimension A_i at the j^{th} level ($j \geq 1$), attribute A_{i+1} is chosen for the partition or classification at the same level of the concept hierarchy of A_{i+1}. The partition process on each attribute A_is is cycled based on the function such as: $(i+1)$ **mod** D for a given $j < height(ch(A_i))$ till it is stopped.

4. After each cycle of partitioning on attribute A_is, the algorithm will start a new cycle on the attribute A_is and perform further specialized classification based on the set of elements at the next level of the concept hierarchy such that: the level $j \leq height(ch(A_i))$.

5. As soon as the partitioning or classifying is completed at the current level of its corresponding concept hierarchy along the attribute or dimension, the relevant information about the quantitative rule is stored correspondingly in a node in terms of multidimensional data structure. The multidimensional data structure used here is similar to the *k-d* tree [5], *k-d-b* tree [19], and other multidimensional trees [20].

6. The algorithm will recursively partition and will not stop the process until either the threshold values are not satisfied or the partition or classification reaches all the bottom levels of the concept hierarchies. The set of threshold values $\mathbf{T}_i$s is defined and can be used to control the depth in a tree or the number of levels of rules. If the threshold values are not used, then partition process will derive the complete set of rules along each dimension corresponding to each level of the concept hierarchies.

7. At each level of classification or partition along attribute A_i, a set of quantitative rules can be derived and the information about the set of derived rules is stored correspondingly in a node $T_{i,j,k}$ at each level of the tree. The number of derived rules depends on the number of elements at the corresponding level of the concept hierarchy. The watershed, boundary range information, and the number of tuples in that range are stored at each level of the tree.

4.2 Definitions for s_{weight} and c_{weight} in Quantitative Rules

The partitioning process is to classify the data tuples in multidimensional space according to the given set of concept hierarchies. The quantitative rule information can be calculated along the partitioning process and stored in the multidimensional tree as defined in **Definition 3.7**.

In general, a derived quantitative rule can be expressed as follows:

$$\forall(x) target\_class(x) \leftrightarrow conditions(x)[s : w, c : w'] \quad (4)$$

The symbol '$\leftrightarrow$' indicates that $conditions(x)$ are both the sufficient and necessary conditions for the derived rules [9].

The value of s_{weight} indicates the percentage of data tuples in a partitioned node satisfying the conditions of the rule out of the total population in R:

$$s_{weight} = \frac{\sum_{\forall x \in p_{i,j,k}} count(x)}{card(R)} \quad (5)$$

where

1. $p_{i,j,k}$ is the k^{th} class (partition) from the dimension $(i-1)$ with respect to $h_{i,j,k}$ at the j^{th} level of the knowledge tree for attribute A_i.

2. $p_{i,j,k}$ is the partition derived from the subtree at the level of $(i-1)$. For any given node or subtree t_{i-1}, there might be more than one partitions derived from t_{i-1} that belong to $h_{i,j,k}$.

3. card(R) is the total number of data tuples in the data set.

The value of c_{weight} describes percentage of data tuples satisfying the conditions of the rule out of all the tuples in the partitions derived from the same subtree root.

$$c\_weight = \frac{\sum_{\forall x \in p_{i,j,k}} count(x)}{\sum_{\forall k} \sum_{\forall x \in p_{i,j,k}} count(x)} \quad (6)$$

The value of $\sum_{\forall x \in p_{i,j,k}} count(x)$ is the total number of tuples that belong to the k^{th} class (partition) at the j^{th} level of the knowledge tree for attribute A_i. The partition $p_{i,j,k}$ contains k classifications or partitions from the same subtree.

The value of $\sum_{\forall k} \sum_{\forall x \in p_{i,j,k}} count(x)$ is the total number of tuples in these k partitions $p_{i,j,k}$s derived from the subtree with respect to $h_{i,j,k}$.

5 Parallel Partitioning Algorithm

The rule derivation process is goal-driven and attribute-oriented data partition with the assistance of the set of concept hierarchies. As soon as the partition of data tuples for classification is done at each level, each partitioned set of data tuples is independent. The partitioning process at the next level can be done in a parallel fashion, and so on in further deep level. The process of the rule derivation can speed up.

Given a set of predetermined order for the set of attributes participating in the partitioning process, the parallel version of the partitioning algorithm scans the data and classifies the knowledge along each dimension or attribute by dimension cycling and concept hierarchy descending. As soon as the partitioning is done at each dimension, the next partitioning can be performed parallelly. Each processor P_I receives a partitioned set of data tuples, and performs further partitioning based on corresponding concept hierarchy as rule derivation process. At the end of partitioning process, the quantitative rule information is derived and stored. Data tuples in partitioned sets from current partitioning process can be treated as subtasks and sent to other available processors for further processor in parallel.

5.1 Parallel Top-Down Partition

Assume that there are N processors $P_1, P_2, ..., P_N$, and the corresponding algorithm as well as the input to and output from the algorithm is described in **Algorithm 5.1** .

5.2 Control Mechanism in Parallel Top-Down Partitioning

Each processor P_I in the free pool is always waiting for the message sent to. As soon as it finishes the current partition, the processor will send K messages making corresponding K procedure calls with the k partitioned groups $r(R)_{i,j,1}$, $r(R)_{i,j,2}$, ..., $r(R)_{i,j,k}$ and release itself. When each processor P_I is released from the current execution of the procedure call, it will start a cycle by waiting for a new message sent to it. When the number of free processors P_Is is greater then the number of partitioned groups K, there is no procedure call with a partitioned group $r(R)_{i,j,k}$ that needs to wait for processor to be released. Otherwise, the procedure call with a partitioned group $r(R)_{i,j,k}$ will wait for a released processor. Since a processor will release itself as soon as the end of execution of the current procedure, the called procedure that waits for the processor will always be assigned to a newly released one.

5.3 The Cost Analysis for Parallel Top-Down Partition Algorithm

Since the parallel partitioning at each level is employed, the partitioning cost at each level will be determined by the following:

Input: The universal database schema R
The set of tuples $r(R)_{i,j,k}$ in $T_{i,j,k}$.
The set of concept hierarchies CH
The ordered set of cardinalities CARD
The specification of a learning task
The set of attributes A_is $\in$ R - TA such that:
$| A_i | \leq | A_{i+1} |$ and $| ch_{i,j} | \leq | ch_{i+1,j} |$
The set of threshold values $\tau = \{ \tau_1, ..., \}$

Output: A set of learning quantitative rules R

Algorithm 5.1 Parallel Top-Down Partition

begin

1. *processor P_I receives $r(R)_{i,j,k}$ and partitions the set of tuples in $r(R)_{i,j,k}$ in $T_{i,j,k}$ for given i, j, and k with $h_{i,j,k}$.*

2. *compute the values of i and j for the partitioning at the next level of the tree $T_{i,j,k}$ by* **Equation** *2 and 3:*

   repeat
   if $(i+1)$ **mod** $D < i$ **mod** D &
   $j < height(c_{i+1})$ **then**
   $j = j + 1$ */* start a new dimension cycle */*
   endif
   $i = (i+1)$ **mod** D
   until $j \leq height(ch(A_i))$

3. *generate a set of newly partitioned groups $r(R)_{i,j,1}$, $r(R)_{i,j,2}$, ..., $r(R)_{i,j,k}$ based on the computed i and j.*

4. *count the number of tuples in each $h_{i,j,k}$.*

5. *compute the s_{weight} and c_{weight} in each partitioned group $r(R)_{i,j,1}$, $r(R)_{i,j,2}$, ..., $r(R)_{i,j,k}$.*

6. *store the corresponding derived rule related information for each partitioned group in each corresponding tree node $T_{i,j,k}$.*

7. **if** $i < D$ *and* $j < max(height(c_i))$ **then**

   (a) *generate and initialize all the child nodes for each $T_{i,j,k}$ for the partition in the next cycle.*

   (b) *Assign the set of partitioned group $r(R)_{i,j,1}$, $r(R)_{i,j,2}$, ..., $r(R)_{i,j,k}$ to the set of processors $P_{1,}, P_{2,}, ..., P_k$, and generate k calls to* **Parallel Top-Down Partition.**

   endif

end

$$O(\mathbf{max}(i/o\_time(r(R)_{i,j,k}) + w\_time(r(R)_{i,j,k}) + p\_time(r(R)_{i,j,k}))) \quad (7)$$

where w_time is the time for the process assigned with $r(R)_{i,j,k}$ to wait for a released processor, p_time is the processing time to partition $r(R)_{i,j,k}$, and

$$i/o\_time = \lceil \frac{\mid r(R)_{i,j,k} \mid \times sizeof(tuple(R))}{B} \rceil \quad (8)$$

The total cost of the parallel partition process from all the levels is:

$$O(\sum_{l=1}^{L} \mathbf{max}(i/o\_time(r(R)_{i,j,k}) + w\_time(r(R)_{i,j,k}) + p\_time(r(R)_{i,j,k}))) \quad (9)$$

where $L = 1, ..., \sum_{i=0}^{D} i * height(ch_i)$.

Although **Algorithm 5.1** independently partitions the set of tuples according to the given set of concept hierarchies in parallel, the load allocated among each processor may not be balanced since the number of tuples in each partitioned set that corresponds to $h_{i,j,k}$ in a $ch_{i,j}$ is not necessarily equalized. In order to resolve this issue, we will present the following algorithm that will allow each processor to work not only in independence but also in balanced load.

6 Load Balance Parallel Partitioning

Assume that there are N processors $P_1, P_2, ..., P_N$, and the corresponding algorithm as well as the input to and output from the algorithm is described in **Algorithm 6.1**.

6.1 Load Balance Partitioning Algorithm in Parallelism

Algorithm 6.1 is based on MPI (message passing interface) and SPMD model (single processor multiple data), and can be divided into two parts: master processor P_0 and set of slave processors $\{P_1, P_2, ..., P_N - 1\}$.

Each processor $P_I, 1 \leq I \leq N-1$ is assigned with equal load from $r(R)$ such as:

$$b_I = \lceil \frac{\mid r(R) \mid}{(N-1)} \rceil$$

Each processor $P_I, 1 \leq I \leq N-1$ maintains a local tree T_I that corresponds to the set b_I with derived rule information. Processor P_0 maintains a global tree **T** that corresponds to r(R) with derived rule information. All the trees in $P_I, 0 \leq I \leq N-1$ have the identical structures with same height, dimension cycle, levels, etc.

Initial, P_0 sends all b_Is to each $P_I, 1 \leq I \leq N-1$. As soon as each $P_I, 1 \leq I \leq N-1$ receives the equally partitioned data set b_I, it starts local partition process in cycles $1 \leq j \leq \mathbf{max}(height(CH))$. At the end of each cycle, each processor $P_I, 1 \leq I \leq N-1$ sends P_0 the corresponding derived rule information stored in local T_I. When P_0 receives set of derived information in T_Is from each $P_I, 1 \leq I \leq N-1$ for a given cycle or level j, it will update the tree **T** on P_0 accordingly.

6.2 Cost Analysis for Load Balance Partitioning Algorithm

The analysis will be based the cost on the both P_0 and $P_I, 1 \leq I \leq N-1$. For each processor $P_I, 1 \leq I \leq N-1$, the knowledge tree T_I and its associated derived rule information are stored in the main memory. The initial construction of the knowledge tree needs to retrieve the data set from the secondary memory, and there are input/output costs involved.

Since the data tuples in $r(b_I)$ only need to be partitioned based on their corresponding classifications or categories, the partition process at each level for each attribute can be done in a single pass of table scan. For each partition process, a list of hash buckets in the main memory can be allocated corresponding to the number of partitions at each of concept hierarchy. Each bucket is used to store the partitioned tuples belonging to that classification or partition. The data tuples in each hash bucket will be flushed into the secondary memory when the bucket is full. Double buffering techniques can be used to reduce the contention and to improve the performance. For each pass of table scan, it is necessary to read all the data tuples in the data set $r(b_I)$ from the secondary memory, so I/O costs in the partition process will be dominant by the number of data blocks fetched as follows:

$$\frac{\mid r(b_I) \mid \times sizeof(tuple(R))}{B} \quad (10)$$

where $\mid b_I \mid$ is the number of tuples in the data set b_I, $sizeof(tuple(R))$ is the size of a tuple in terms of bytes in R, and B is the size of a data block.

Since each slave processor works independently on each equally partitioned set of data, the total I/O cost of the partition process in each partition cycle for each $P_I, 1 \leq I \leq N-1$ will be bound by the following:

$$I/O_{I,j} = O(\lceil \frac{\mid r(b_I) \mid \times sizeof(tuple(R))}{B} \rceil \times \mid C_j \mid \quad (11)$$

The cost involed in the partioning processing for each $P_I, 1 \leq I \leq N-1$ can be summerized as:

$$Cost_{I,j} = I/O_{I,j} + Wait_{I,j}$$

where $Wait_{I,j}$ is the cost for P_0 to wait for receiving $T_{I,j}$ from each $P_I, 1 \leq I \leq N-1$.

We assume the set of received information in each $T_{I,j}$ can be buffered and queued, and also I/O cost is

Algorithm 6.1 Load Balance Partitioning Algorithm

begin

if P_I*'s rank* $== 0$ **then** {

for P_I; $I = 1, ..., N_P - 1$ **pardo**
send $r(b_I)$ *to* P_I *such that:* $\mid r(b_I) \mid = \lceil \frac{\mid r(R) \mid}{(N-1)} \rceil$
endfor

for $j = 1, ..., \mathbf{max}(height(CH))$ **do**
for P_I; $I = 1, ..., N_P - 1$ **pardo**
receive the derived rule information in each $T_{I,j}$ *w.r.t.* C_j.
expand and update the tree $\mathbf{T}$ *with each* $T_{I,j}$ *from each* P_I.
endfor
endfor

}
else { /* P_I*'s rank* $= 1, ..., N_P - 1$ */

1. *receive* b_I *from* P_0
2. *initialize local tree* T_I
3. **for** $j = 0$; $j++$; $j < \mathbf{max}(height(CH))$ **do**
for $i = 0$; $i++$; $i <$ $\mid C_j \mid$ **do**
if $\mathbf{T}_i \leq \tau_i$ & $j < height(ch_i)$ **then**

(a) *partitioning the data tuples in* b_I *along the dimension* i *for attribute* A_i, *and generating* k *partitioned groups* $r_{i,j,k}(R)$ *based on the corresponding to concept hierarchy* $ch(A_i)$, *where* $k = 1, 2, ...$ *for the given* i *and* j.
/* *partition and group the tuples according to the values of* $h_{i,j,k} \in ch_{i,j}$ */

(b) *count the number of tuples in each* $h_{i,j,k}$.

(c) *compute the* s_{weight} *and* c_{weight} *for each partitioned group.*

(d) *store the corresponding derived rule related information for each partitioned group in each tree node* $T_{i,j,k}$ *in the local tree.*

(e) *generate and initialize all the child nodes of* $T_{i,j,k}$ *for partitioning in the next cycle.*

endif
endfor
send the derived rule information in T_j *w.r.t* C_j *to* P_I.
endfor

end

dominant comparing to CPU cost. So the cost to build the tree $\mathbf{T}$ on processor P_0 side is ignored.

The total I/O cost for the entire partitioning processing is as follows:

$$O(\sum_{j=1}^{|\mathbf{C}|} \mathbf{max}(Cost_{1,j}, Cost_{2,j}, ..., Cost_{N-1,j}) \quad (12)$$

where $\mid \mathbf{C} \mid$ is the number of dimension cycles in the partitioning processing.

Since P_0 sends $N-1$ messages, and each $P_I, 1 \leq I \leq N-1$ sends one message in each dimension cycle, the total number of messages sent during the execution of **Algorithm 6.1** is: $(N-1) + (N-1) \times \mid \mathbf{C} \mid = (N-1) \times (\mid \mathbf{C} \mid +1)$.

7 Discussions and Summary

Based on serial incremental algorithm, we presented two parallel algorithms for top-down quantitative rule derivation. These two algorithms have not been implemented yet. Although **Algorithm 6.1** was proposed as load balance oriented, we are planning to implement **Algorithm 6.1** and provide further study and analysis from performance point of view to validate this property.

Because the partitioning approaches presented are guided by a set of concept hierarchies, each partitioned set of data tuples is independent to each other. As soon as the partitioning is done at the current dimension and/or current level of the concept hierarchy, the further partitioning in the next step can be done in a parallel fashion. This property makes the incremental algorithm scalable to perform rule derivation process in paralllel.

From analytical point of view, **Algorithm 6.1** improves **Algorithm 5.1** in terms of total number of messages sent during parallel rule derivation process. The total number of messages sent in **Algorithm 6.1** is: $(N-1) + (N-1) \times \mid \mathbf{C} \mid = (N-1) \times (\mid \mathbf{C} \mid +1)$ instead of $\prod_{j=1}^{|\mathbf{C}|} \prod_{i=1}^{|C_j|} \mid h_{i,j,k} \mid$.

Since the multidimensional tree data structure is employed to store the derived quantitative rule information, the partition topology of the tree in terms of multidimensional data space, corresponds to the given set of concept hierarchies. As for the invariant concept hierarchies, the partitioned topology of the multidimensional space is invariant. From this, the set of data tuples within each partition corresponds to a concept element or branch in a concept hierarchy. Since internal nodes of the tree are stored in the main memory, the set of quantitative rules stored in the multi-level tree can be retrieved without extra I/O cost. The set of quantitative rules stored in the main memory can be used for OLAP process. Further, the quantitative information about the set of derived rules can be updated within the

main memory whenever a database undergoes updates [4, 22].

8 Acknowledgments

The author would like to express appreciation to the anonymous referees for their valuable comments and suggestions. The author would like to express sincere thanks to Dean Edward Lieblein, the faculty and staff at the School of Computer and Information Sciences, Nova Southeastern University. Special thanks to Dr. Bell Selvaraj.

References

[1] R. Agrawal and J. C. Shafer. Parallel mining of association rules. In *IEEE Transactions on Knowledge and Data Engineering,* Volume 8, Number 6, pp. 962-969, December, 1996.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the* 20^{th} *International Conference on Very Large Data Bases*, pp. 487-499, August, 1994.

[3] R. Agrawal, T. Imielinski, A. Swarmi. Mining associations between sets of items in massive databases. In *Proceedings of 1993 ACM SIGMOD International Conference on Management of Data*, pp. 207-216, May, 1993.

[4] N. F. Ayan, A. U. Tansel, and E. Arkun. An efficient algorithm to update large itemsets with early prunning. In *Proceedings of the* 5^{th} *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 287-291, August, 1999.

[5] J. L. Bentley. Multidimensional binary search trees in database applications. In *IEEE Transactions on Software Engineering*, Volume 5, Number 4, pp. 333-340, July 1979.

[6] M. S. Chen, J. Han, and P. S. Yu. Data mining: an overview from a database perspective. In *IEEE Transactions on Knowledge and Data Engineering*, Volume 8, Number 6, pp. 866-883, 1996.

[7] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy. *Advances in Knowledge Discovery and Data Mining, (eds).* Morgan Kaufmann Publishers, 1996.

[8] J. Han and Y. Fu. Mining multiple-mevel association rules in large databases. In *IEEE Transactions on Knowledge and Data Engineering*, Volume 11, Number 5, pp. 798-804, 1999.

[9] J. Han, Y. Cai, and N. Cercone. Data-driven discovery of quantitative rules in relational databases. In *IEEE Transactions on Knowledge and Data Engineering*, Volume 5, Number 1, pp. 29-40, 1993.

[10] J. Han, Y. Cai, and N. Cercone. Knowledge discovery in databases: an attribute-oriented approach. In *Proceedings of the* 18^{th} *Very Large Data Bases Conference*, pp. 547-559, August, 1992.

[11] B. Liu, W. Hsue, and Y. Ma. Integrating classification and association rule mining. In *Proceedings of* 4^{th} *International Conference on Knowledge Discovery and Data Mining*, pp. 80-86, August, 1998.

[12] H. Mannila, H. Toivonen, and A. I. Verkamo. Efficient algorithms for discovering association rules. In *Proceedings of AAAI Workshop on Knowledge Discovery in Databases*, pp.181-192, 1994.

[13] Ryszard S. Michalski, Ivan Bratko, and Miroslav Kubat. *Machine Learning and Data Mining: Methods and Applications*(edited). J. Wiley, 1998.

[14] R. S. Michalski, J. G. Carbonell, and Mitchell, *Machine Learning, An Artificial Intelligence Approach*, Volume 2, Morgan Kaufmann Publishers, 1986.

[15] R. S. Michalski. A theory and methodology of inductive learning. In *Michalski et. al. (eds.), Machine Learning: An Artificial Intelligence Approach*, Volume 1, Morgan Kaufmann Publishers,pp. 83-134, 1983.

[16] R. Motwani, E. Cohen, M. Datar, S. Fujiware, A. Gionis, P. Indyk, J. Ullman, and C. Yang. Finding interesting associations without support pruning. In *Proceeddings of the* 16^{th} *International Conference on Data Engineering,* pp. 489-500, February, 2000.

[17] J. S. Park, M. S. Chen, and P. S. Yu. An effective hash-based algorithm for mining association rules. In *Proceedings of 1995 ACM SIGMOD International Conference on Management of Data*, pp. 175-186, May, 1995.

[18] J. R. Quinlan. *C4.5: Programs for Machine Learning*, Morgan Kaufmann Publishers, 1993.

[19] J. T. Robinson. The k-d-b tree: a search structure for large multidimensional dynamic index. *Proceedings of 1981 ACM SIGMOD International Conference on Management of Data*, pp. 10-18, 1981.

[20] T. Sellis, N. Roussopoulus, and C. Faloutsos. The r^+-tree: a dynamic index for multi-dimensional objects. In *Proceedings of the* 13^{th} *International Conference on Very Large Data Bases*,pp. 507-518, August, 1987.

[21] R. Srikant and R. Argaral. Mining quantitative association rules in large relational tables. In *Proceedings of 1996 ACM SIGOMD International Conference on Management of Data*, pp. 1-12, June, 1996.

[22] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. An efficient algorithm for incremental updation of association rules in large databases. In *Proceedings of the* 3^{rd} *International Conference on Knowledge Discovery and Data Mining*, pp. 263-266, August, 1997.

[23] M. Zaki and C. T. Ho. *Large-Scale Parallel Data Mining* (edited), Lecture Note in Computer Science, Volume 1759, Springer Verlag, 2000.

[24] M. J. Zaki, C. T. Ho, and R. Agrawal. Scalable parallel classification for data mining on shared-memory multiprocessors. In *Proceedings of* 15^{th} *IEEE International Conference on Data Engineering*, pp 198-205, March 1999.