PROCEEDINGS

Workshop on Large−Scale Parallel KDD Systems

in conjunction with the

5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD99)

Edited by

Mohammed J. Zaki
Rensselaer Polytechnic Institute, Troy, NY

Ching−Tien (Howard) Ho
IBM Almaden Research Center, San Jose, CA

August 15
San Diego, CA

Workshop on Large−Scale Parallel KDD Systems

August 15th, 1999
San Diego, CA

in conjunction with the
5th ACM SIGKDD International Conference on
Knowledge Discovery and Data Mining (KDD99)

Workshop Chairs

Mohammed J. Zaki
Rensselaer Polytechnic Institute

Ching−Tien (Howard) Ho
IBM Almaden Research Center

Program Committee

David Cheung, University of Hong Kong, Hong Kong
Alok Choudhary, Northwestern University
Alex A. Freitas, PUC−PR, Brazil
Robert Grossman, University of Illinois−Chicago
Yike Guo, Imperial College, UK
Hillol Kargupta, Washington State University
Masaru Kitsuregawa, University of Tokyo, Japan
Vipin Kumar, University of Minnesota
Reagan Moore, San Diego Supercomputer Center
Ron Musick, Lawrence Livermore National Lab
Srini Parthasarathy, University of Rochester
Sanjay Ranka, University of Florida
Arno Siebes, CWI, Netherlands
David Skillicorn, Queens University, Canada
Paul Stolorz, Jet Propulsion Lab
Graham Williams, CSIRO, Australia

Foreword

With the unprecedented rate at which data is being collected today in almost all fields of human endeavor, there is an emerging economic and scientific need to extract useful information from it. Many companies already have data warehouses in the terabyte range (e.g., FedEx, UPS, Walmart, etc.). Implementation of data mining ideas in high−performance parallel and distributed computing environments is thus crucial for ensuring system scalability and interactivity.

The goal of this workshop is to bring researchers and practitioners together in a setting where they can discuss the design, implementation, and deployment of large−scale, parallel KDD systems, which can manipulate data taken from very large enterprise or scientific (e.g., space missions, human genome project, etc.) databases, regardless of whether the data are located centrally or are globally distributed.

These informal proceedings contain 8 papers that were accepted for presentation at the workshop. In addition, the program includes four invited presentations by Reagan Moore from San Diego Supercomputer Center, Umeshwar Dayal from Hewlitt−Packard Research, Graham Williams from Commonwealth Scientific and Industrial Research Organisation, Australia, and by Robert Grossman and Yike Guo from the University of Illinois, Chicago, and Imperial College, UK. Finally, the program includes a panel on the future of large−scale parallel data mining, with the panelists including Vipin Kumar (U. Minnesota), Ron Musick (Lawrence Livermore National Labs), and Foster Provost (Bell Atlantic).

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 and Howard Ho

Workshop Chairs

Workshop on Large−Scale Parallel KDD Systems

August 15th, 1999
San Diego, CA

i**n conjunction with the**
5th ACM SIGKDD International Conference on
Knowledge Discovery and Data Mining (KDD99)

8:40am	Opening Remarks
8:45−9:25am	Invited Talk: **Collection−Based Data Management** Reagan Moore, San Diego Supercomputer Center, USA
9:25−10:15am	Session I: **Mining Frameworks**
	A high performance implementation of the data space transfer protocol (DSTP) S. Bailey, E. Creel, R. Grossman, S. Gutti, H. Sivakumar , University of Illinois−Chicago, USA
	Active data mining in a distributed setting S. Parthasarathy, S. Dwarkadas, M. Ogihara, University of Rochester, USA
10:15−10:30am	Coffee Break
10:30−11:10am	Invited Talk: **Large−Scale Data Mining Applications: Requirements and Architectures** Umeshwar Dayal, Hewlitt−Packard Corp., USA
11:10−12:00am	Session II: **Association Rules**
	Parallel branch−and−bound graph search for correlated association rules S. Morishita, A. Nakaya, University of Tokyo, Japan
	Parallel algorithms for mining association rule mining on large scale PC Cluster T. Shintani and M. Kitsuregawa, University of Tokyo, Japan
12:00−2:00pm	Lunch Break

2:00−2:40pm	Invited Talk: **Integrated Delivery of Large−Scale Data Mining Systems** Graham Williams, CSIRO, Australia
2:40−3:30pm	Session III: **Clustering and Sequences**
	A data clustering algorithm on distributed memory machines S. Dhillon and D. S. Modha , IBM Almaden Research Center, USA
	Parallel sequence mining on SMP machines M. Zaki, Rensselaer Polytechnic Institute, USA
3:30−3:45pm	Coffee break
3:45−4:25pm	Invited Talk: **Communicating Data Mining: Issues and Challenges in Wide Area Distributed Data Mining** Bob Grossman and Yike Guo, University of Illinois−Chicago, and Imperial College, UK
4:25−5:15pm	Session IV: **Classification**
	Efficient parallel classification using dimensional aggregates S. Goil and A. Choudhary , Northwestern University, USA
	Learning rules from distributed data L.O. Hall, N. Chawla, K.W. Bowyer, and W. P. Kegelmeyer, University of South Florida and Sandia National Labs., USA
5:15−6:15pm	Panel: **Large−Scale Data Mining: Where is it Headed?** Vipin Kumar, University of Minnesota Ron Musick, Lawrence Livermore National Labs Foster Provost, Bell Atlantic Mohammed Zaki (moderator), Rensselaer Polytechnic Institute
6:15pm	Closing Remarks

A High Performance Implementation of the Data Space Transfer Protocol (DSTP)

S. Bailey, E. Creel, R Grossman,* S. Gutti, and H. Sivakumar
National Center for Data Mining (M/C 249)
851 S. Morgan Street
University of Illinois at Chicago
Chicago, IL 60607

Abstract

With the emergence of high performance networks, clusters of workstations can now be connected by commodity networks (meta-clusters) or high speed networks (super-clusters) such as the very high speed Backbone Network Service (vBNS) or Internet2's Abilene. Distributed clusters are enabling a new class of data mining applications in which large amounts of data can be transferred using high performance networks and statistically and numerically intensive computations can be done using clusters of workstations.

In this paper, we briefly describe a protocol called the Data Space Transfer Protocol (DSTP) for distributed data mining. With high performance networks, it becomes possible to move large amounts of data for certain queries when necessary. This paper describes the design of a high performance DSTP data server called Osiris which is designed to efficiently satisfy data requests for distributed data mining queries. In particular, we describe 1) Osiris's ability to lay out data by row or by column, 2) a scheduler intended to handle requests using standard network links and requests using network links enjoying some type of premium service, and 3) a mechanism designed to hide latency.

1 Introduction

In this paper we consider some of the issues that arise in distributed data mining when large amounts of data are moved between sites. One of the fundamental tradeoffs in distributed data mining is between the cost of computation and the accuracy of results. We assume: 1) that there is a cost for moving data between sites, and 2) that the most accurate model is obtained by moving all the data to a single site. Leaving some or all of the data in place, building local models, and merging the resulting models, produces a model which is less accurate, but which, in general, is also less expensive to compute.

The cost of moving data to a central location with the commodity Internet has tended to produce either distributed data mining systems which build local classifiers and then combine them or data mining systems that use standard interfaces such as ODBC or JDBC. These protocols work best when moving relatively small amounts of data to a central location. Examples of the former include JAM [18] and BODHI [15]; examples of the latter include Kensington [11].

In a previous paper [22], we have pointed out that there are many intermediate cases in which building classifiers that are close to the optimal one results in moving some of the data, leaving some of the data in place, building local classifiers, and combining them. In this paper, we are concerned with the design of network protocols and middle-ware for distributed data mining systems which have the ability to move some data and to leave other data in place. For example, Papyrus [7] is a distributed data mining system of this type.

Three fundamental challenges faced by distributed data mining systems are:

> *Problem A.* How can the analysis of distributed data be simplified?
>
> *Problem B.* How can the amount of data per site be scaled?
>
> *Problem C.* How can the number of sites be scaled?

To address Problem A, we introduced a protocol called the Data Space Transfer Protocol (DSTP) [1]. In this paper, we are concerned with how we can design DSTP data servers for distributed data mining which scale up as the amount of data per site increases (Problem B). We describe a high performance DSTP server we are designing called Osiris, which is a component of a distributed and high performance data mining system we are building called Papyrus [7].

One method of satisfying the computing and i/o requirements for high performance data mining is to use clusters of workstations [7] [16] [19] — *compute*

clusters to satisfy the CPU requirements and *data clusters* to satisfy the i/o requirements. With the recent advances in high performance networks, geographically distributed clusters of workstations can be connected not only with commodity networks but also with high performance networks such as the NSF vBNS Network supported by MCI and the Internet2 Abilene Network supported by Qwest. For example, for the distributed data mining tests reported below, we used a data cluster in Chicago connected to a compute cluster in Washington, D.C. over a DS-3 link running at 45 Mb/s. Our first DSTP implementation provided approximately 3 Mb/s of throughput, while our current implementation provides approximately 30 Mb/s of throughput, a 10x improvement. See Table 1.

Based upon our previous experience analyzing the performance of another distributed data mining system we built [10], we decided to focus on three questions:

What do we store? More precisely, how should we physically layout the data on the server? By row or by column? Can we precompute intermediate quantities to speed up queries?

What do we move? More precisely, to what extent should data or meta-data be moved from node to node? There are several possibilities: we can move data, we can move predictive models, or we can move the results of computations. If we decide to move data, we can move data by table, by row, or by column.

How do we move it? What application protocol should be used for moving data in data space? How can data be moved in parallel between nodes? How can QoS be exploited to improve data transport? What is the effect of latency on data mining queries? What transport protocol should we use? Given multiple requests to a data server, how should the requests be scheduled?

The 10x performance gain we mentioned above resulted from careful understanding of these issues. Section 2 describes related work and background material. Section 3 describes data space and the data space transfer protocol. Section 4 describes the DSTP server and three experimental studies. Section 5 is the conclusion and summary.

2 Background and Related Work

In this section, we provide some background material and discuss some of the related work in this area. With the exception of [19] and [16], the work we know of in this area is limited to data mining over commodity networks. This section is adapted from [8].

Several systems have been developed for distributed data mining. Perhaps the most mature are: the JAM system developed by Stolfo et. al. [18], the Kensington system developed by Guo et. al. [11], and BODHI developed by Kargupta et. al. [15]. These systems differ in several ways:

Data strategy. Distributed data mining can choose to move data, to move intermediate results, to move predictive models, or to move the final results of a data mining algorithm. Distributed data mining systems which employ *local learning* build models at each site and move the models to a central location. Systems which employ *centralized learning* move the data to a central location for model building. Systems can also employ *hybrid learning*, that is, strategies which combine local and centralized learning. JAM and BODHI both employ local learning while Kensington implements a centralized approach using standard protocols such as JDBC to move data over the commodity Internet.

Task strategy. Distributed data mining systems can choose to coordinate a data mining algorithm over several sites or to apply data mining algorithms independently at each site. With *independent learning*, data mining algorithms are applied to each site independently. With *coordinated learning,* one (or more) sites coordinate the tasks within a data mining algorithm across several sites.

Model Strategy. Several different methods have been employed for combining predictive models built at different sites. The simplest, most common method is to use *voting* and combine the outputs of the various models with a majority vote [4]. *Meta-learning* combines several models by building a separate meta-model whose inputs are the outputs of the various models and whose output is the desired outcome [18]. *Knowledge probing* considers learning from a black box viewpoint and creates an overall model by examining the input and the outputs to the various models, as well as the desired output [11]. Multiple models, or what are often called ensembles or committees of models, have been used for quite a while in (centralized) data mining. A variety of methods have been studied for combining models in an ensemble, including Bayesian model averaging and model selection [17], partition learning [6], and other statistical methods, such as mixture of experts [23]. JAM employs meta-learning, while Kensington employs knowledge probing.

Papyrus is designed to support different data, task and model strategies. For example, in contrast to JAM, Papyrus can not only move models from node to node, but can also move data from node to node, when that strategy is desired. In contrast to BODHI, Papyrus is built over a data warehousing layer which can move data over both commodity and high performance networks. Also, Papyrus is a specialized system which is designed for clusters, meta-clusters, and super-clusters, while

JAM, Kensington and BODHI are designed for mining data distributed over the Internet.

Moore [16] stresses the importance of developing an appropriate storage and archival infrastructure for high performance data mining and discusses work in this area. The distributed data mining system developed by Subramonian and Parthasarathy [19] is designed to work with clusters of SMP workstations and like Papyrus is designed to exploit clusters of workstations. Both this system and Papyrus are designed around data clusters and compute clusters. Papyrus also explicitly supports clusters of clusters and clusters connected with different types of networks.

3 Data Space and the Data Space Transfer Protocol

We begin by describing some of the key concepts following [1].

Data Space. We assume that data is distributed over nodes in a global network, which we call a *data space*.

Rows and Columns. Although the data may be more complicated, we assume that the data is organized into tables, and that each table is organized into rows (records) and columns (observations). Records may contain missing values.

Catalog Files. Each DSTP server has a special file called the *catalog file* containing meta-data about the data sets on the server.

DSTP. We assume that there is a server on each node which can move data to other nodes using a protocol called the *data space transfer protocol (DSTP)*. Depending upon the request, DSTP servers may return one or more columns, one or more rows, or entire tables. DSTP servers can also return meta-data about tables and the data they contain.

Universal Correlation Keys. A row may have one or more keys. Certain keys called *Universal Correlation Keys* (UCK) are used for relating data on two different DSTP servers. For example, key-value pairs (k_i, x_i) on DSTP Server 1 can be combined with key-value pairs (k_j, y_j) on DSTP Server 2 to produce a table (x_k, y_k) in a DSTP client. The DSTP client can then find a function $y = f(x)$ relating x and y.

Since DSTP client applications need only collect meta-data from the catalog files and need only move the relevant columns, these type of applications tend to scale better as the number of sites increases (Problem C) than distributed data mining applications which must move entire files. Recall that we are interested in the case in which some data is moved. Of course, if sufficient accuracy can be obtained by local analysis followed by combining models, this is usually less expensive than strategies which require that data be moved.

Notice that from this perspective, distributed databases are concerned with the efficient *updates* of distributed *rows*, while distributed data mining applications are concerned with the efficient *reading* and analysis of distributed *columns*.

In the next section, we describe our efforts to produce DSTP servers which can efficiently manage large data sets.

4 The Osiris DSTP Server

Osiris is a high performance DSTP Server which is designed to provide efficient read access to data. In our design, efficient read access is delivered by implementing high performance storage support, high performance network transfer support, and differentiated service support.

In this section we discuss our implementations of these support mechanisms and some preliminary experimental results which attempt to quantify the relative performance gains for each technique. All three mechanisms are implemented in process space and do not require any special tuning of the underlying hardware or operating systems. We felt it was important to provide performance improvements that were independent of the underlying system in order to increase portability.

4.1 Rows and Columns

Tabular data may be laid out on disk by row or by column. Since data from disk is transfered by block, certain queries will be more efficient when the data is laid out by row (*horizontally*) and other queries will be more efficient when the data is laid out by column (*vertically*).

DSTP client applications accessing data may request either rows of data or columns of data. If a column of data is requested and the underlying storage layout is *horizontal*, then each block will contain quite a bit of unwanted data. The same is true if a row of data is requested and underlying layout is *vertical*.

Since horizontal layouts speed up certain distributed data mining queries and vertical layouts speed up others, Osiris stores data in both formats. Although this doubles the amount of space required, the I/O traffic is reduced significantly. Since Osiris is a distributed system, the I/O traffic ultimately passes through a network communication link. Since network bandwidth is sufficiently more expensive than disk

Horizontal Store: Store Size = 4.4 GB				
NC	ADR in Mb/s	TDT in Giga bytes	TTT in seconds	EPR in Events/second
1	3.06	4.4	11777.5	64
2	6.07	4.4	5926.59	253
4	10.05	4.4	3590.40	655
8	16.92	4.4	2132.05	2811
16	23.32	4.4	1550.91	7731
32	34.93	4.4	1032.24	23245

Vertical Store: Store Size = 4.0 GB				
NC	ADR in Mb/s	TDT in Mega bytes	TTT in seconds	EPR in Events/second
1	1.39	269.5	1549.05	400
2	2.75	269.5	797.00	1554
4	3.81	269.5	566.42	4377
8	6.75	269.5	320.45	15482
16	9.74	269.5	223.05	44590
32	13.96	269.5	152.52	126918

Table 1: Performance analysis of horizontal vs. vertical stores.
NC - Number of clients requesting data
ADR - Aggregate Data Rate
TDT - Total Data Transferred
TTT - Total Time Taken for completion of application
EPR - Events Processing Rate

capacity, we feel the 2X increase in required storage is more than compensated for.

The following results are from a proof of concept DSTP data server located at the University of Illinois at Chicago being accessed by multiple clients located at an Internet2 member facility in Washington, D.C. called Highway One. The two sites are connected by the NSF/MCI vBNS Network. Even though vBNS is an OC-3 network offering maximum bandwidth of 155 Mb/s, the end nodes at Highway One were connected via a DS-3 link, which limited the maximum bandwidth of the testbed to 45 Mb/s.

For this test, we used an application benchmark we developed called the *Event Benchmark*, which is broadly derived from high energy physics. The data consists of a large collection of *events*, with each event containing several hundred attributes. An energy like function is computed from the attributes of an event and the energies of the event are histogrammed.

To better understand quantitative effects of the Horizontal/Vertical Layout strategy, we first laid out the data *horizontally* and ran the application, and then we laid out the data *vertically* and ran the application.

In the first case, all the event data was stored as rows (i.e., each event was a row). In the second case, the event data was stored attribute by attribute as columns. The *Event Benchmark* specifies that event level summary data is to be stored separately and analyzed at run-time to find out which attributes are to be requested and processed. In other words, this particular application requests columns of data based on some criteria. Therefore, we expected that a *vertical* layout should provide better performance.

Table 1 shows the performance results. The Event Processing Rate (EPR) is an an application benchmark of efficiency. Aggregate Data Rate (ADR) and Total Data Transferred (TDT) are system performance measures. The desired result is to maximize application efficiency with the least load on the system. Clearly, the *vertical* layout provided better performance, as expected.

This experiment demonstrates the effect that layout has on application performance. Because we cannot predict whether applications will request rows or columns, storing the data both *horizontally* and *vertically* will guarantee performance gain.

4.2 Differentiated Service Support with Diff-Serv Scheduler

Osiris is being developed to simultaneously serve clients on both commodity and high performance networks. Because of the *premium* nature of high performance networks, it is desirable that clients on these networks have some precedence over clients on commodity net-

NC	ART-P in seconds	ART-PS in seconds	ART-C in seconds	ART-CS in seconds
1	606.3	570.4	422.1	447.4
2	577.5	557.5	445	463.5
3	574.4	558.5	568.5	581.3
4	566.6	566.5	715	740
5	565.9	562.16	880.9	892.7

Table 2: Performance of Diff-Serv scheduler
NC - Number of clients requesting data per service type
ART-P - Average run-time for premium clients (no scheduling)
ART-PS - Average run-time for premium clients (with Diff-Serv scheduling)
ART-C - Average run-time for commodity clients (no scheduling)
ART-CS - Average run-time for commodity clients (with Diff-Serv scheduling)

works. Treating *premium clients* and *commodity clients* differently constitutes a type of Quality of Service(QoS) called *differentiated services* [20].

Differentiated service support in Osiris is another mechanism that attempts to contribute to the requirement of efficient read access. In this context, efficiency refers to system wide resource utilization as opposed to per process performance.

Because the currently popular Internet protocol suite (IP) does not support any kind of QoS mechanism, we chose to implement differentiated service support as a characteristic of the scheduling mechanism for client requests to Osiris. We refer to this scheduler as the Diff-Serv Scheduler.

When a client attaches to Osiris, it informs the server whether it is a *premium client* or a *commodity client*. Data block requests are then scheduled for service as they arrive with *premium client* requests getting preferential treatment. Please see [12] for full design and implementation details of the Diff-Serv Scheduler.

For our experimental study, a single server was run on a machine connected to the network through Switched Fast Ethernet (100 Mbps). An equal number of clients connected to both Switched Fast Ethernet (*premium clients*) and Switched Ethernet (*commodity clients*) were launched and connected to the server.

The *premium clients* each made 10,000 random block requests, and the *commodity clients* each made 5,000 random block requests. The default block size for Osiris is 16KB. Every client waited for an exponentially distributed random delay between block requests. This delay was introduced to cause a Poisson distribution of request arrivals to the server and was an attempt to simulate real application behavior.

Experiments were conducted which compare system performance with Diff-Serv scheduling turned on against system performance with Diff-Serv scheduling turned off. Measurements were made with a total of two to ten clients. The results are presented in Table 2. Please note that when Diff-Server scheduling is turned off, the system defaults to FIFO scheduling.

The desired results were achieved. In all cases, *premium client* response time improved while *commodity client* response time diminished when our implementation of Diff-Serv scheduling was turned-on.

4.3 High Performance Network Transfer Support with PSocket

It has been well documented that latency characteristics of TCP over wide area networks, or more precisely networks with large "bandwidth · delay" products, have a significant negative impact on per process communication performance [13]. Various protocol and implementation level solutions have been suggested [14] [5]. One technique is for the sender to send multiple messages to the receiver in parallel [21].

In order to provide high performance network transfer support, we allow a single process to break up a message and then send the pieces in parallel over multiple communication links (e.g., TCP sockets) to the receiver who then rebuilds the entire message. We refer to this technique as Transport Layer Multiplexing and have implemented a simple-to-use interface for application integration called PSocket (as in Parallel Socket). For full details please refer to [2].

Osiris will use PSockets to increase the data transfer rate on a per client process basis. The results in Table 3 are from a single, non-threaded sender process using PSocket, located at the University of Illinois at Chicago, sending data to a single non-threaded receiver process using PSocket located at Highway One. The two sites are connected by the NSF/MCI vBNS Network. Highway One connects to vBNS via a DS-3 link, which limited the maximum theoretical bandwidth of the testbed to 45 Mb/s.

The experiment measured the wall clock transfer time of a 100 MByte buffer. Results show that with a PSocket of size 5, a large portion of the practical

TM	TT in seconds	AATR in Mbps
traditional single socket	96	8.3
PSocket size 2	57	14.0
PSocket size 3	34	23.5
PSocket size 4	30	26.7
PSocket size 5	26	30.8
PSocket size 6	26	30.8
PSocket size 7	26	30.8

Table 3: Performance of Transport Layer Multiplexing with PSocket. (Note: The practical limit of the 45 Mb/s DS-3 appears to be about 35 Mb/s.)
TM - Transport Mechanism
TT - Transfer Time for 100 MBytes
AATR - Application Apparent Transfer Rate

transfer rate of the DS-3 was consumed by the transfer. As a reference, the transfer rate using traditional, single socket programming was given.

5 Conclusion

In general, the less data which distributed data mining systems move, the less expensive the computation. However, due to the level of accuracy required or to the nature of the data, it is sometimes necessary to move large amounts of data between sites. With the emergence of high performance networks this becomes practical in many circumstances in which it would have previously been impractical.

In this paper, we have described some of the design considerations for a high performance data server called Osiris which is part of the Papyrus distributed data mining infrastructure and presented some experimental results describing its use on an application benchmark requiring the computation of histograms.

In particular, we describe a design which supports high performance storage, high performance network transfer, and differentiated network services, such as commodity links and high performance links. This design provides at least a 10x improvement over a more naive design. We expect this to grow to 100x for certain applications and network configurations.

References

[1] S. Bailey, E. Creel, and R. L. Grossman, DataSpace: Protocols and Languages for Distributed Data Mining, National Center for Data Mining/Laboratory for Advanced Computing Technical Report, http://www.ncdm.uic.edu, 1999.

[2] S. Bailey, R. L. Grossman, Transport Layer Multiplexing with PSocket, National Center for Data Mining Technical Report, 1999.

[3] P. Chan and H. Kargupta, editors, Proceedings of the Workshop on Distributed Data Mining, The Fourth International Conference on Knowledge Discovery and Data Mining New York City, 1999, to appear.

[4] T. G. Dietterich, Machine Learning Research: Four Current Directions, AI Magazine Volume 18, pages 97-136, 1997.

[5] D. J. Farber, J. D. Touch, An Experiment in Latency Reduction, IEEE Infocom, Toronto, June 1994, pp. 175-181.

[6] R. L. Grossman, H. Bodek, D. Northcutt, and H. V. Poor, Data Mining and Tree-based Optimization, Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, E. Simoudis, J. Han and U. Fayyad, editors, AAAI Press, Menlo Park, California, 1996, pp 323-326.

[7] R. L. Grossman, S. Bailey, S. Kasif, D. Mon, A. Ramu and B. Malhi, The Preliminary Design of Papyrus: A System for High Performance, Distributed Data Mining over Clusters, Meta-Clusters and Super-Clusters, Proceedings of the Workshop on Distributed Data Mining, The Fourth International Conference on Knowledge Discovery and Data Mining New York City, August 27-31, 1998, to appear.

[8] R. L. Grossman and Y. Guo, An Overview of High Performance and Distributed Data Mining, submitted for publication.

[9] R. L. Grossman, S. Bailey, A. Ramu and B. Malhi, P. Hallstrom, I. Pulleyn and X. Qin, The Management and Mining of Multiple Predictive Models Using the Predictive Modeling Markup Language (PMML), Information and Software Technology, 1999.

[10] The Terabyte Challenge: An Open, Distributed Testbed for Managing and Mining Massive Data Sets, Proceedings of the 1998 Conference on Supercomputing, IEEE.

[11] Y. Guo, S. M. Rueger, J. Sutiwaraphun, and J. Forbes-Millott, Meta-Learnig for Parallel Data Mining, in *Proceedings of the Seventh Parallel Computing Workshop,* pages 1-2, 1997.

[12] S. Gutti, A Differentiated Services Scheduler for Papyrus DSTP Servers, Master's Thesis, University of Illinois at Chicago, 1999.

[13] V. Jacobson, Congestion Avoidance and Control, SIGCOMM '88, Stanford, CA., August 1988.

[14] V. Jacobson, and R. Braden, TCP Extensions for Long-Delay Paths, RFC-1072, LBL and USC/Information Sciences Institute, October 1988.

[15] H. Kargupta, I. Hamzaoglu and B. Stafford, Scalable, Distributed Data Mining Using an Agent Based Architecture, in D. Heckerman, H. Mannila, D. Pregibon, and R. Uthurusamy, editors, *Proceedings the Third International Conference on the Knowledge Discovery and Data Mining,* AAAI Press, Menlo Park, California, pages 211-214, 1997.

[16] R. W. Moore, C. Baru, R. Marciano, A. Rajasekar, and M. Wan, Data-Intensive Computing, Ian Foster and C. Kesselman, editors, The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann, San Francisco, 1999, pages 105-129.

[17] A. E. Raftery, D. Madigan, and J. A. Hoeting, 1996. Bayesian Model Averaging for Linear Regression Models. Journal of the American Statistical Association 92:179- 191.

[18] S. Stolfo, A. L. Prodromidis, and P. K. Chan, JAM: Java Agents for Meta-Learning over Distributed Databases, *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining,* AAAI Press, Menlo Park, California, 1997.

[19] R. Subramonian and S. Parthasarathy, An Architecture for Distributed Data Mining, to appear.

[20] B. Teitelbaum and J. Sikora (1998), Differentiated Services for Internet2, Draft Proposal, http://www.internet2.edu/qos/may98Workshop/html/diffserv.html

[21] J. D. Touch, Parallel Communication, *the proceedings of Infocomm 1993*, San Francisco CA. March 28-April 1, 1993.

[22] R. L. Grossman and A. Turinsky, Optimal Strategies for Distributed Data Mining using Data and Model Partitions, submitted for publication.

[23] Xu, L.; and M.I. Jordan, M. I. 1993. EM Learning on A Generalised Finite Mixture Model for Combining Multiple Classifiers. In Proceedings of World Congress on Neural Networks. Hillsdale, NJ: Erlbaum

Active Mining in a Distributed Setting *

S. Parthasarathy, S. Dwarkadas, M. Ogihara

Department of Computer Science
University of Rochester
Rochester, NY 14627–0226

{srini,sandhya,ogihara}@cs.rochester.edu

Abstract

Most current work in data mining assumes that the data is static, and a database update requires re-mining both the old and new data. In this article, we propose an alternative approach. We outline a general strategy by which data mining algorithms can be made *active* — i.e., maintain valid mined information in the presence of user interaction and database updates. We describe a runtime framework that allows efficient caching and sharing of data among clients and servers. We then demonstrate how existing algorithms for four key mining tasks: Discretization, Association Mining, Sequence Mining, and Similarity Discovery, can be re-architected so that they maintain valid mined information across i) database updates, and ii) user interactions in a client-server setting, while minimizing the amount of data re-accessed.

1 Introduction

As we enter the digital information era, one of the greatest challenges facing organizations and individuals is how to turn their rapidly expanding data stores into accessible knowledge. Digital data sources are ubiquitous and encompass all spheres of human endeavor: from a supermarket's electronic scanner to a world wide web server, from a credit card reader to satellite data transmissions. Organizations and individuals are increasingly turning to the extraction of useful information, referred to as data mining, from such databases. Such high-level patterns, or inferences, extracted from the data may provide information on customer buying patterns, on-line access patterns, fraud detection, weather trends, etc.

*This work was supported in part by NSF grants CDA–9401142, CCR–9702466, CCR–9705594, CCR-9701911, CCR-9725021, and INT-9726724; DARPA grant F30602-98-2-0133; and an external research grant from Compaq.

A typical data mining technique can be thought of as an exploration over a huge user-defined pattern space, with the role of the data being to select patterns with a desired degree of confidence. The set of accepted patterns is a function of both the data and the user-defined pattern space (controlled by the input parameters). The data mining process tends to be interactive and iterative in nature, i.e., after observing the results from the first round of mining, the user may choose to repeatedly modify the input parameters, thereby affecting the set of accepted patterns. Also, since businesses are constantly collecting data, the data is also subject to change, again affecting the set of accepted patterns.

Most current techniques, which tend to be static in nature, simply re-execute the algorithm in the case of data updates or user interaction. There are several limitations to this approach. First, although many of these techniques have parallel solutions [20, 28, 4] that are efficient in storage, access, and scale, they are still computationally expensive. Second, re-executing the algorithm requires re-examining both the old and new data, and hence I/O continues to be a bottleneck. These problems are further exacerbated in applications such as electronic commerce and stock sequence analysis, where it is important to execute the query in real or near-real time, in order to meet the demands of on-line transactions. Also, more and more such applications are being deployed as client-server applications where the server is physically separate from the client machine. Such a setup is also common within an organization's intra-net when there may be several groups mining, perhaps with separate agendas, from a common dataset. Ensuring reasonable response times in such applications is made more difficult due to the network latency and server load overheads. This leads to the following challenge:

In order to meet the demands of such interactive applications, can existing algorithms be re-architected, making them efficient in the presence of user interactions and data updates, in a distributed client-server setting?

1.1 Proposed Solution

We refer to algorithms that maintain valid mined information in the presence of user interaction and database updates as *active* algorithms. The main challenge is to perform the active mining in a storage and time efficient manner. This paper describes a general strategy by which data mining tasks can be re-architected to work efficiently with the constraints outlined above.

We accomplish our objective by maintaining a *mining summary structure* across database updates and user interactions. On a database update, the revamped algorithm replaces accesses to the old data with accesses to the mining summary structure whenever possible. This ensures that information that is previously mined can be re-used when the database is updated. On a user interaction, the hope is that the mining summary structure can answer the query without accessing the original data.

The design criteria for such a summary structure are: i) it should allow for incremental maintenance as far as possible, i.e., the mining summary structure from the old data along with the data update should ideally be sufficient to produce the new summary structure, avoiding accesses to the old data as far as possible, ii) it should store sufficient information to address a wide range of useful user interactions, and iii) it should be small enough to fit in memory so that accessing it rather than the old data provides a significant performance gain.

While the above solution can potentially solve the active mining problem, deploying these algorithms efficiently in a distributed setting is non-trivial. In typical client-server applications, the client makes a request to the server, the server computes the result, and then sends the result back to the client. The query execution time is significantly influenced by the speed of the client-server link as well as the server load. Since the interactions in our applications are often iterative in nature, caching the aforementioned summary structure on the client side so that repeated accesses may be performed locally eliminates overhead and delays due to network latency and server load. In order for this to be an effective solution, the summary structure should not be very large (re-emphasizing point iii) above), and an efficient mechanism for communicating updates is required.

Such summary structure sharing requires efficient caching support. We have built a general-purpose framework called InterAct that facilitates the development of interactive applications. InterAct supports sharing among interactive client-server applications. The key to the framework is an efficient mechanism to facilitate client-controlled consistency and sharing of objects among clients and servers (this allows applications that can tolerate some information loss to take advantage of this to increase efficiency by reducing communication). Advantages within the scope of our work include: the ability to cache relevant data on the client to support interactivity, the ability to update cached data (when the data changes on the server) according to application or user preferences while minimizing communication overhead, and the ability to extend the computation boundary to the client to reduce the load on the server. We use this framework to develop our applications.

1.2 Contributions

In this paper:

1. We describe a general methodology for creating *active* mining solutions for existing applications.
2. We present active mining solutions for discretization, association mining, sequence mining, and similarity discovery in a distributed setting.
3. We describe the InterAct framework, which, along with the changes to the algorithms for active mining, allows effective client-server distribution of the computation.

The next section presents the InterAct framework on top of which we implement our active mining algorithms. We also outline our general approach to the problem of making algorithms *active*. Sections 3 (Discretization),4 (Association/Sequence Mining), and 5 (Similarity Discovery) describe the applications we look at and our specific approach to make each of them *active*. We present and evaluate our experimental results in Section 6. Section 8 details our conclusions.

2 InterAct Framework

InterAct is a runtime framework that presents the user with a transparent and efficient data sharing mechanism across disparate processes. The goal is to combine efficiency and ease-of-use. InterAct allows clients to cache relevant shared data locally, enabling faster response times to interactive queries. Further, InterAct provides flexible client-controlled mechanisms to map and specify consistency requirements for shared data.

In order to accomplish its goal of transparently supporting interactive applications, InterAct:

- Defines a data (structure) format for shared objects that is address space and architecture independent. Our implementation relies on the use of C++ and some programmer annotation to identify the relevant information about shared data to the runtime framework.
- Identifies, defines, and supports the different types of consistency required by such applications

(described in [16]). In many domains (electronic commerce, credit card transactions), the data that is being queried is constantly being modified. The rate at which these modifications are required to be updated in the client's cached copy may vary on the basis of the domain, application, or specific user. This rate can be controlled by exploiting the appropriate consistency model to enhance application performance.

- Provides an underlying mechanism to transparently handle the consistency demands as well as complex object transfer requirements. The goal here is to reduce programming complexity by hiding as much of the underlying communication from the application developer as possible.

For more framework details and the consistency types supported, see [16].

2.1 Active Mining and InterAct

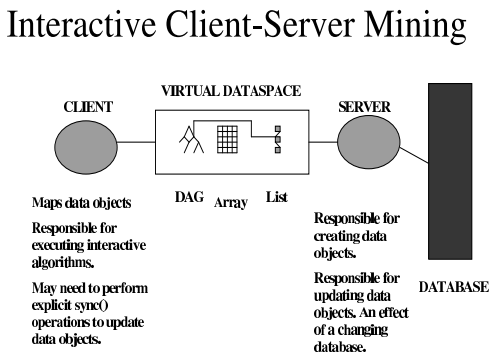


Figure 1: Client-Server Mining using InterAct

Since mining applications typically operate on large data sets that reside on a data server, communicating these datasets to the client would be infeasible. However, in this paper we show that it is possible to design useful summary structures for a range of mining applications, so that subsequent queries can operate on these summary structures. This summary data structure can be generated by the data distiller (server), and subsequently operated on by the client. Hence, the applications can be structured as shown in Figure 1, so that the data server is responsible for creating the data structure(s), mapping them onto a *virtual shared dataspace*, and subsequently keeping them up-to-date. The client can then map the data structure(s) from the *virtual shared dataspace* under a desired consistency model, thus enabling the client to respond to interactive queries without having to go to the data server.

In order for the above setup to be effective the summary data structure should satisfy three key properties. First, it should be able to directly answer a range of interactive queries without requiring access to the data as far as possible. This criterion minimizes the client-server communication, as well as server load. Second, the summary structure should be incrementally maintainable. This criterion ensures that changes to the data can be rapidly reflected in the summary structure. Third, the summary structure should not be very large as otherwise communicating it to the client may be very expensive. In the ensuing sections, we describe how such summary structures can be designed for Discretization, Association and Sequence Mining, and Similarity Discovery.

3 2-D discretization

Discretization is the process by which the range of a given *base* attribute (or independent variable) is partitioned into mutually exclusive and exhaustive intervals based on the value of a related *goal* attribute (or dependent variable), whose behavior we would like to predict or understand. Discretization has primarily been used for summarization [8], as well as for growing a decision tree [19].

Typically, a single attribute is used as the decision variable. However, one can also consider extensions to more than one base attribute (e.g., $X > 5 \wedge Y < 6$) as long as the decisions remain simple. The need for this is often encountered where repetitive applications of the single-attribute discretization do not provide optimal results, while a single, integrated two-dimensional approach does. In [17], the partitioning of a two dimensional base attribute space is defined in terms of *control point*s. A single control point partitions the base attribute space into 4 rectangular regions. The rectangular regions are induced by drawing lines through the control point that are perpendicular to the two axes. Two control points partition the base attribute into up to 9 regions. The effect of discretization is to approximate the behavior of the goal attribute as the same for all points in a region. The purpose of the algorithm is to find the position of the control point(s) that optimizes a given objective function.

The input to a discretization algorithm could be either the raw data or a joint probability density function (pdf) of the base and goal attributes derived from the data. Using the data directly eliminates errors associated with pdf estimation. However, using a pdf enables one to use more meaningful error metrics such as Entropy [9]. Second, it permits users to encode domain knowledge by altering the shape and type of kernel (normal, poisson, binomial, etc.) used for density estimation. Further, it lends itself to a client-server architecture where density

estimation could be done on the server and a compact representation shipped to the client.

Evaluating any pair of base attributes involves 2 steps: computing the three dimensional probability density (pdf) estimate (two base attributes and the goal attribute), and searching for the optimal (determined by the objective function: Classification Error) discretization.

3.1 Interactivity

The idea in interactive discretization is that an end user ought to be able to modify process parameters. The interactive features currently supported in our algorithm include: i) choosing from a set of algorithms (brute force search, approximate search), and metrics (entropy, error), to compute the optimal discretization, ii) changing the number of control points (1, or 2), iii) changing the position of control points, and iv) changing the parameters for pdf estimation.

Ideally, all these features need to be supported efficiently without excessive I/O or computation.

3.2 Summary Structure

As mentioned earlier, the summary structure required to support such interactions efficiently is the joint pdf $p(base_1, base_2, goal)$. This pdf is estimated at discrete locations. While several techniques exist to estimate the density of an unknown pdf, the most popular ones are histogram, moving window, and kernel estimates [7]. We use the histogram estimate described in [7]. The advantage of this estimate is that it can be incrementally maintained in a trivial manner (a histogram estimate is essentially the frequency distribution normalized to one). Moreover, the more complicated kernel estimates can easily be derived from this basic estimate [7].

With a few modifications, the histogram pdf estimate can handle each of the interactions described above. If there are n points in the estimate, computing the objective function for a given control point takes $O(n)$ time. Since the objective of discretization is to find the control point(s) that optimizes the given objective function, a brute force search will take $O(n^2)$ ($O(n^4)$ for two control points) time. Recently, we have shown that smarter methods can reduce this time complexity to $O(n)$ ($O(n^2)$ for two control points) with additional memory ($O(n)$). Alternatively, fast approximate searches like simulated annealing can be used to generate good discretizations quickly [17].

Changing the control point in small incremental ways (moving the control point along one of the two axes in unit steps), enables one to compute the new objective functions (entropy or error) in unit time at the cost of additional storage ($O(n)$). Changing the parameters for pdf estimation can also be done quickly using the histogram estimate [7].

In order for the summary structure to be cost effective, the size of the structure (n, the number of points in the estimate) must be small enough so that i) it can be cached easily on remote clients, and ii) it can allow for faster interactions. In the next section, we describe an experiment that explores the tradeoff between the size of the summary structure and the accuracy of the discretization obtained.

3.2.1 Accuracy vs. Summary Size

We evaluate the premise that it is possible to create a condensed representation of the data (probability density function) without serious degradation in the quality of discretizations obtained. This enables efficient client-server partitioning, and allows off-loading parts of the computation to another machine, thereby reducing the load on the server, and potentially improving interaction efficiency.

Experimentally, we have found that the number of points at which the pdf needs to be evaluated (determined by grid size) without significant quality degradation is not large. Our results are summarized in Figure 2. The results reported are for two synthetic data sets, XOR and LL. These are described in [17] [1]. Both have 100,000 instances and 2 base attributes. It is easy to see that the quality (error minimization) of discretization does not improve much beyond a grid size of 64^2.

Data Set	Grid Size	Error
XOR	16^2	18.87%
XOR	32^2	18.13%
XOR	64^2	17.90%
XOR	128^2	17.86%
LL	16^2	12.47%
LL	32^2	12.2%
LL	64^2	12.2%
LL	128^2	12.3%

Figure 2: Effect of grid size for pdf evaluation on quality of results

The resulting summary structure for discretization satisfies all the properties for efficient active mining: it is bounded and small, it can be used to handle a wide range of client queries without going back to the original database, and it can be incrementally maintained.

4 Association/Sequence Mining

In this section, we consider two of the central data-mining tasks, i.e., the discovery of association rules

[1] Available via anonymous ftp from ftp.cs.rochester.edu/pub/u/srini

and the discovery of sequences. The discussion below follows [28] (associations) and [27] (sequences).

The problem of mining association rules over *basket* data was introduced in [2, 3]. It can be formally stated as: Let $\mathcal{I} = \{i_1, i_2, \cdots, i_m\}$ be a set of m distinct attributes, also called *items*. Each transaction T in the database $\mathcal{D}$ of transactions, has a unique identifier, and *contains* a set of items, such that $T \subseteq \mathcal{I}$. An *association rule* is an expression $A \Rightarrow B$, where $A, B \subset \mathcal{I}$, are sets of items called *itemsets*, and $A \cap B = \emptyset$. Each itemset is said to have a *support* S if $S\%$ of the transactions in $\mathcal{D}$ contain the itemset. The association rule is said to have *confidence* C if $C\%$ of the transactions that contain A also contain B, i.e., $C = S(A \cup B)/S(A)$, i.e., the conditional probability that transactions contain the itemset B, given that they contain itemset A.

Sequence Mining can be thought of as association mining over temporal datasets. A *sequence* is an ordered (over time) list of nonempty itemsets. A sequence of itemsets $\alpha_1, \ldots, \alpha_n$ is denoted by $(\alpha_1 \mapsto \cdots \mapsto \alpha_n)$. The *length* of a sequence is the sum of the sizes of each of its itemsets. The database is divided into a collection of customer sets where each customer set contains the set of transactions that customer is involved in in order of occurrence. For a database D and a sequence α, the *support* or *frequency* of α in D, is the number of customers in D whose sequences contain α as a subsequence. A rule $A => B$ involving sequence A and sequence B is said to have *confidence* c if $c\%$ of the customers that contain A also contain B.

The basic approach to mining associations and sequences is a two step iterative approach. First, identify the set of candidate associations/sequences for a given number of items. Second, compute the set of associations/sequences from the candidate set that meet the user-specified criteria, forming the basis for the candidates in the next iteration (adding one to the number of items considered). We use the ECLAT [28] (associations) and SPADE [27] (sequences) algorithms as the basis for our work.

4.1 Interactivity

The idea in interactive association mining (or interactive sequence mining) is that an end user be allowed to query the database for association rules at differing values of support and confidence. The goal is to allow such interaction without excessive I/O or computation. Interactive usage of the system normally involves a lot of manual tuning of parameters and resubmission of queries that may be very demanding on the memory subsystem of the server. In most current algorithms, multiple passes have to be made over the database for each $< support, confidence >$ pair. This leads to unacceptable response times for online queries. Our approach to the problem of supporting such queries efficiently is to create pre-processed summaries that can quickly respond to such online queries.

A typical set of queries that such a system could support include: i) **Simple Queries:** identify the rules for support x%, confidence y%, ii) **Refined queries:** where the support value is modified ($x + y$ or $x - y$) involves the same procedure, iii) **Quantified Queries:** identify the k most important rules in terms of support, confidence pairs or find out for what support/confidence values can we generate exactly k rules, iv) **Including Queries:** find the rules including itemsets $i_1, \ldots, i_n$, v) **Excluding Queries:** compute the rules excluding itemsets $i_1, \ldots, i_n$, and vi) **Hierarchical Queries:** treat items $i_1, \ldots, i_n$, as one item and return the new rules.

4.2 Summary Structure

In [1], the concept of an Association Lattice L is defined. An association X is said to be adjacent to an association Y if one of them can be obtained from the other by adding a single item. Specifically, an association X is a parent of Y if Y can be obtained by adding one item to X. We allow directed edges from parents to children. It is then easy to see that if a directed path exists from a vertex V to a vertex U then $V \subset U$. Further, each node in the lattice is weighted by the support S of the given association it represents. The sequence lattice is obtained in a similar manner.

The preprocessing step of the algorithm involves computing such a lattice for a small enough support S_{min}, such that all future queries will involve a support S larger than S_{min}. If the above holds, and given such a lattice, we can produce answers to all but one (**Hierarchical queries**) [2] of the queries described in the previous section at interactive speeds without going back to the original database. This is easy to see as all of the queries will basically involve a form of pruning over the lattice. A lattice, as opposed to a flat file containing the relevant associations/sequences, is an important data structure as it permits rapid querying for associations [1] and sequences [18]. This lattice can also be incrementally maintained for associations [25] and sequences [18]. Due to limited space, we do not describe it here.

4.2.1 Accuracy vs. Summary Size

Unlike in discretization, the size of the summary structure is not bounded for a choice of S_{min}. It depends on the data and the choice of S_{min}. For small enough S_{min} the lattice can be very large (larger than the original data itself in some cases!). However, for most practical cases (e.g., $S_{min} = 0.05\%$, dataset

[2] These queries require recomputation on the server. However, because of the way we access the data, and the way it is stored we limit accesses to the old data.

= 170MB) the resulting lattice (4MB) is manageable and the applications can benefit from client side caching of the data structure.

The summary structure satisfies the three properties for efficient active mining: it can be used to handle a wide range of client queries without going back to the original database, it can be incrementally maintained, and for most practical instances, the size of the lattice is not too large.

5 Similarity Discovery

Similarity is a central concept in data mining. Discovering the similarity among attributes enables reduction in dimensions of object profiles as well as provides useful structural information on the hierarchy of attributes. Das et al [6] proposed a novel measure for attribute similarity in transaction databases. The similarity measure proposed compares the attributes in terms of how they are individually correlated with other attributes in the database. The choice of the other attributes (called the probe set) reflects the examiner's viewpoint of relevant attributes to the two. Das et al show that the choice of the probe set strongly affects the measurement.

There are some limitations to this basic approach. First, when the examiner does not know what the relevant attributes are, their approach offers no solutions. A brute force search would be impractical. Second, the approach limits the probe elements to singleton attributes and does not allow boolean attribute formulae.

If one is not interested in probe attributes of small frequency, an alternative approach can be to use the associations generated by an algorithm such as ECLAT [28] as the probe set. The similarity metric between attributes "a" and "b" can be defined in terms of association sets (A, the set of all associations involving "a" but excluding "a". For instance if "adl" were a valid association, then "dl" would belong to the set A. Similarly, B, the set of all associations involving "b" but excluding "b".) and their associated supports (sup):

$$Sim(A,B) = \frac{\sum_{x \in A \cap B} \max\{0, 1 - \alpha |\sup_{\mathcal{A}}(x) - \sup_{\mathcal{B}}(x)|\}}{\|A \cup B\|},$$

where α is a user-defined normalizing variable that defaults to one. This approach is fast and scales well in practice. It also permits boolean attribute formulae(a limitation of the Das et al [6] approach) as part of the probe set. Since we use associations as the probe set, this approach can also be used to measure the similarity between different homogeneous datasets and is not limited to measuring attribute similarity.

5.1 Interactive Similarity

In our approach the following interactions are currently supported: i)**Boolean Pruning:** Prune the probe space (association sets) to only those parts of the association sets that satisfy a given boolean formula, ii) **Identifying influential attributes:** Identify the (set of) probe attribute(s) that contribute most to the similarity/dissimilarity metric, and iii) **Changing the minimum support**.

5.2 Summary Structure

In this application, the summary structure required is the association lattice described in Section 4.2. For dataset similarity, the association lattices of both datasets are required.

Once the association lattices are obtained, the basic algorithm computes the similarity measure. The different interactions are supported as follows.

For Boolean Pruning, the algorithm basically prunes both lattices according to the boolean formula, yielding sets A^1 and B^1. The similarity metric is then recomputed by replacing A with A^1 and B with B^1. For Identifying Influential Attributes, for a singleton attribute "l", the algorithm prunes out all elements in the association lattice that do not contain "l". It then computes the similarity between the two datasets. This step is repeated for all singleton attributes and the resulting similarities are sorted. The higher ranked attributes influence similarity while the lower ranked attributes influence dis-similarity.

5.2.1 Accuracy vs. Summary Size

Like association mining, the size of the summary structure is not bounded for a choice of S_{min}. However, unlike association mining, similarity discovery is less attuned to this choice. Fixing an S_{min} apriori is an acceptable solution for the purpose of computing similarities. This limits the size of the data structure, ensuring that the three properties for active mining are satisfied for this application.

6 Experimental Evaluation

In order to completely evaluate all aspects of our work, we evaluate the impact of our summary structure with respect to three qualities; its interactive performance, its incremental performance, and the efficacy of client-server work distribution by caching the summary structure and executing queries locally. We first describe in detail the queries we evaluated on each of the applications, and their associated datasets.

6.1 Application Properties

We executed a series of queries for each application. For association mining, we executed a simple query (find rules with support x%) followed by a quantified query (find the 400 most important associations). For sequence mining, we executed a combination of including (all sequences including item x) and excluding sequences (all sequences excluding item y). For discretization, we executed the base algorithm

(find the optimal discretization) and then asked the system to move the control points to a new location and compute the new error. In similarity discovery, we asked the system to compute the pairwise similarities for four datasets and then asked it to recompute the similarities under boolean pruning, as well as identify the most influential attributes.

Each of the queries was executed on an appropriate dataset. For association mining, we executed our queries using a synthetic dataset generated adopting the methodology described in [3]. The dataset we used (T10.I6.D3200) contained on average 10 items per transaction, and 3,200,000 transactions. The size of the resulting dataset is 140MB.

For sequence mining, we executed our queries using a synthetic dataset generated by a similar procedure [27]. The dataset we used (C250.I6.T10) contained on average 10 transactions per customer, and 250,000 customers, where each transaction is variable in length. The size of the resulting dataset is 55MB.

For discretization, we used the XOR dataset [17]. The dataset has 100,000 instances and 3 attributes (two base,one goal) per instance. There are 2 categories for the goal attribute, C0 and C1. The size of the resulting dataset is 4MB.

For similarity discovery, we executed our queries against a real dataset, the Reuters dataset [3]. The data set consists of 21578 articles from the Reuters newswire in 1987. Each article has been tagged with keywords. The size of the dataset is 27MB. For our evaluation, we represented each news article as a transaction with each keyword being an item.

For each of the applications considered: Associations, Sequences, Discretization and Similarity, the size of the summary structures were 3.3MB, 1.0MB, 0.5MB, and 2.0MB, respectively. It is easy to see that in all of the cases the summary structure is a significant reduction from the original dataset and is small enough to enable effective *active* mining.

6.2 *Active* Mining Performance

We summarize the results on interactive and incremental mining performance here. In this paper, we have described a methodology for off-loading the interactive querying feature onto client machines as opposed to executing on the server, and shipping the results to the data mining client. In order to clearly demonstrate the effectiveness of this approach, we wanted to compare executing queries on slower clients (143Mhz and 270Mhz UltraSparcs) using the designed summary structure versus recomputing the result on the fastest server we have available (600MHz Alpha Station 4100s).

The results of the experiment are shown in Table 1. The first column corresponds to the application we evaluated, the second column contains the execution time of the query on a 143Mhz client using the appropriate summary structure, the third column similarly contains the execution time on a 270Mhz client, and the fourth column represents the execution time of running the query from scratch on the 600 MHz Alpha, without the use of the summary structure. For all of the applications, the execution time with the summary structure is orders of magnitude faster than re-executing the query from scratch. This is despite the fact that the results obtained for re-executing the query without the summary structure is on a much faster server.

The fifth column of our table represents the speedup obtained from maintaining the structure incrementally, rather than re-creating it on a database update. This part of the experiment was also performed on the 600MHz Alpha Station 4100. The (*incr* 5%) in the column header corresponds to the increment size. The datasets described in the previous sections are divided into two partitions, one containing 95% of the transactions (or instances) and the other containing the remaining 5%. The first partition we assume is the original dataset, while the second partition is treated as the increment dataset. The speedup numbers in this column compare the speedup of using an incremental algorithm as opposed to re-executing the algorithm on the entire (original + increment) data (column 4). The performance gains from the incremental approach ranges from good (speedup of 7) to excellent (speedup of 18). As expected, incrementally maintaining the summary structure for the discretization application results in the best speedup since it is the easiest to maintain.

6.3 Distributed Performance

In typical client-server applications, the client makes a request to the server, the server computes the result, and then sends the result back to the client. Since the interactions in our applications are often iterative in nature, caching the summary data structure on the client side so that repeated accesses may be performed locally can potentially improve query execution times.

We present results on the efficacy of caching the summary structure in a distributed environment consisting of SUN workstations connected by 10 or 100 Mbps switched Ethernet. The clients in each application interact with the server by sharing the summary data structures with the server. The server creates the summary data structure and updates it corresponding to changes in the database (which we simulate). The potential gain from client-side caching depends on a number of factors: the size of the shared data, the speed of the client, the network latency, the server load, and the frequency of data modification. We evaluate the effect of each of these factors.

[3] www.research.att.com/ lewis/reuters21578.html

Application	Client (143Mhz)	Client (270Mhz)	Recompute	Incremental Speedup (*incr* 5%)
Association Mining	2.4	1.5	540.7	10
Sequence Mining	0.58	0.35	150	7
Discretization	0.87	0.55	505.8	18
Similarity	0.35	0.11	10	10

Table 1: Active Mining Performance: Execution Times in seconds

Application	Client(143)					Client(270)				
	CSC	SSRC		L-SSRC		CSC	SSRC		L-SSRC	
Ethernet (Mbps)		10	100	10	100		10	100	10	100
Association	2.4	4.05	1.6	7.2	2.5	1.5	2.5	1.4	5.1	2.3
Sequence	0.58	0.85	0.55	1.35	0.86	0.35	0.63	0.5	1.18	0.73
Discretization	0.87	1.35	0.67	2.75	1.08	0.55	0.94	0.6	1.6	0.98
Similarity	0.35	1.5	0.55	2.7	0.98	0.11	0.9	0.37	2.4	0.94

Table 2: Time (in seconds) to Execute Query in a Distributed Environment

We ran each of our applications under the following scenarios:

1. Client-Side Caching (CSC): the client caches the summary structure and executes the query on the local copy (the execution times reported here do not reflect the time to communicate the summary structure, which gets amortized over several queries).

2. Server Ships Results to Client (SSRC): the client queries the server and the server ships the results back to the client. This scenario is similar to the use of an RPC mechanism. In order to better understand the impact of server load, we varied the number of clients serviced by the server from one (SSRC) to eight (**L**oaded-SSRC).

We measured the time to execute each query under both scenarios. We evaluated each scenario on a range of client machines, from an UltraSparc (143Mhz) machine to an UltraSparc IIi (270Mhz). In each case, our server was an 8-processor 336 MHz UltraSparc II machine. Results are presented in Table 2 for these scenarios under two different network configurations. We varied the network configuration by choosing clients that are connected to the server via a 10 Mbps or a 100 Mbps Ethernet network. For each of the applications considered: Associations, Sequences, Discretization, and Similarity, the size of the results shipped by the server (total data communicated) were 1.5MB, 0.25MB, 0.5MB and 0.75MB respectively.

The results in Table 2 show that client-side caching is beneficial for all but a few of the cases. In particular, the following trends are observed. Client-side caching is more beneficial under the following scenarios: the network bandwidth is low (speedups from client-side caching under the 10Mbps configuration are larger (1.5 to 23) than the 100Mbps numbers (0.6 to 9)), the server is loaded (comparing the L-SSRC column (speedups of 1.1 to 9) with the SSRC column(speedups of 0.6 to 3.5) with a 100 Mbps network), the client is a fast machine (comparing the columns involving the 270Mhz clients versus the 143Mhz clients), or the time to execute the query is low (comparing the row involving the similarity discovery with the row involving association mining). In other words, the benefits from client-side caching are a function of the computation/communication ratio. The lower the ratio, the greater the gain from client-side caching. The fact that InterAct enables such caching is very useful for such applications especially when deployed on the Internet.

In addition, the client maps the shared summary data structures using one of the set of consistency models provided by InterAct. Choosing the right consistency model for a given application depends on its tolerance for stale data. Updates are then transmitted to the client according to the consistency model chosen. Results obtained show that the average update times are several orders of magnitude faster than existing approaches such as RPC. For a detailed analysis of our update protocol and results pertaining to these applications, see [16].

7 Related Work

7.1 Distributed Data Mining Systems

Several systems have been developed for distributed data mining. The JAM [22](Java Agents for Meta-learning) and the BODHI [13] 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 [12] 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 [23] architecture is a general-purpose architecture designed with data mining applications in mind to work with clusters of SMP workstations. Both this system and the Papyrus system [11] are designed around data servers, compute servers, and clients. 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 making existing algorithms *active*.

7.2 Incremental Mining

In [5], an incremental algorithm for maintaining association rules is presented. A major limitation of this algorithm is that it may require $O(k)$ database (original plus increment) scans, where k is the size of the largest frequent itemset. In [10], two incremental algorithms were presented – the *Pairs* approach stores the set of frequent 2-sequences, while the *Borders* algorithm keeps track of the frequent set and the negative border. An approach very similar to the Borders algorithm was also proposed in [25].

There has been almost no work addressing the incremental mining of sequences. One related proposal in [26] uses a dynamic suffix tree based approach to incremental mining in a single long sequence. However, we are dealing with sequences across different customers, i.e., multiple sequences of sets of items as opposed to a single long sequence of items. To the best of our knowledge there has been no work to date on the incremental mining of discretization and similarity discovery.

7.3 Interactive Mining

A mine-and-examine paradigm for interactive exploration of associations was presented in [14]. The idea is to mine and produce a large collection of frequent patterns. The user can then explore this collection by the use of *templates* specifying what's interesting and what's not. They only consider inclusive and exclusive templates (corresponding to our **Including** and **Excluding** queries), whereas our approach handles a wider range of queries, in an efficient manner.

A second approach to exploratory analysis is to integrate the constraint checking inside the mining algorithm. One such approach was presented in [21]. Recently, [15] presented the CAP algorithm for extracting all frequent associations matching a rich class of constraints. Our approach relies on constraining the final results rather than integrating it inside the mining algorithm.

An online algorithm for mining associations at different values of support and confidence, was presented in [1]. Like their approach, we rely on a lattice framework to produce results at interactive speeds. Our approach relies on a different base algorithm [28] for generating associations and this allows us to compute a wider range of queries, as well as, compute such queries faster.

An interactive approach to discretization was presented in [24] for traditional one-dimensional discretization. They also use a probability density estimate of the base attribute to allow for certain user interactions in a manner similar to ours. However, their problem domain is much simpler then ours and therefore the interactive queries supported are relatively easier to compute. We are not aware of any such work on interactive mining, within the domain of sequence and similarity discovery.

Most of the incremental and interactive mining approaches tend to focus on isolated applications leading to a proliferation of solutions with little or no inter-operability. Our approach is the first that tries to integrate the incremental and interactive components in a distributed setting. Furthermore, we outline a general strategy for making mining algorithms *active* in such a setting.

8 Conclusions

In this paper, we described our approach to active data mining in a client-server setting. We presented a general method for creating efficient interactive mining algorithms, and in addition, demonstrated its efficacy in a distributed setting using the InterAct framework. We applied this general methodology to several data mining tasks: discretization, association mining, sequence mining, and similarity discovery.

To summarize our method, we maintain a *mining summary structure* that is valid across database updates and user interactions. On a user interaction, the mining summary structure can answer the query without re-accessing the actual data. On a database update, the amount of the original database that needs to be re-examined is minimized. Lastly, by caching the summary structure on the client using InterAct, we can eliminate overheads due to network latency and server loads.

Experimental results show that executing queries using the appropriate summary structure can improve

performance by several orders of magnitude. Furthermore, for all the applications considered, the summary structures can be incrementally maintained with up to an 18-fold improvement over re-creating the summary structures on a database update. Finally, up to a 23-fold improvement in query execution times was observed when the clients cache the summary structure and execute the query locally.

References

[1] C. Aggarwal and P. Yu. Online generation of association rules. In *IEEE International Conference on Data Engineering*, February 1998.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD Conf. Management of Data*, May 1993.

[3] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Inkeri Verkamo. Fast discovery of association rules. In U. Fayyad and et al, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI Press, Menlo Park, CA, 1996.

[4] 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.

[5] D. Cheung, J. Han, V. Ng, and C. Wong. Maintenance of discovered association rules in large databases: an incremental updating technique. In *12th IEEE Intl. Conf. on Data Engineering*, February 1996.

[6] G. Das, H. Mannila, and P. Ronkainen. Similarity of attributes by external probes. In *Proceedings of the 4th Symposium on Knowledge Discovery and Data-Mining*, 1998.

[7] L. Devroye. A course in density estimation. In *Birkhauser:Boston MA*, 1987.

[8] J. Dougherty, R. Kohavi, and M. Sahami. Supervised and unsupervised discretization of continuous features. *12th ICML*, 1995.

[9] U. Fayyad and K. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. *14th IJCAI*, 1993.

[10] R. Feldman, Y. Aumann, A. Amir, and H. Mannila. Efficient algorithms for discovering frequent sets in incremental databases. In *2rd ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, May 1997.

[11] 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.

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

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

[14] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A. I. Verkamo. Finding interesting rules from large sets of discovered association rules. In *3rd Intl. Conf. Information and Knowledge Management*, pages 401–407, November 1994.

[15] R. T. Ng, L. Lakshmanan, J. Jan, and A. Pang. Exploratory mining and pruning optimizations of constrained association rules. In *ACM SIGMOD Intl. Conf. Management of Data*, June 1998.

[16] S. Parthasarathy and S.Dwarkadas. Shared state for client server applications. TR716, Department of Computers Science, University of Rochester, June 1999.

[17] S. Parthasarathy, R. Subramonian, and R. Venkata. Generalized discretization for summarization and classification. In *PADD98*, January 1998.

[18] S. Parthasarathy, M. Zaki, M. Ogihara, and S. Dwarkadas. Incremental and interactive sequence mining. TR715, Department of Computers Science, University of Rochester, June 1999.

[19] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, Los Altos CA, 1993.

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

[21] R. Srikant, Q. Vu, and R. Agrawal. Mining Association Rules with Item Constraints. In *3rd Intl. Conf. on Knowledge Discovery and Data Mining*, August 1997.

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

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

[24] R. Subramonian, R. Venkata, and J. Chen. A visual interactive framework for attribute discretization. In *Third International Conference on Knowledge Discovery and Data Mining*, pages 82–88, 1997.

[25] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. Incremental updation of association rules. In *KDD97*, Aug 1997.

[26] K. Wang. Discovering patterns from large and dynamic sequential data. *J. Intelligent Information Systems*, 9(1), August 1997.

[27] M. J. Zaki. Efficient enumeration of frequent sequences. In *7th Intl. Conf. on Information and Knowledge Management*, November 1998.

[28] 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*, 1(4):343-373, December 1997.

Large Scale Data Mining Applications: Requirements and Architectures

Umeshwar Dayal, Qiming Chen, Meichun Hsu
Hewlett−Packard Laboratories, 1501 Page Mill Road, MS 1U4, Palo Alto, CA 94303, USA
{dayal, qchen, mhsu}@hpl.hp.com

Abstract

Increasingly, organizations are beginning to mine data stored in very large corporate data warehouses for "business intelligence", for example, profiling customer behaviour, detecting fraud, understanding or predicting market trends, and the like. These business intelligence applications pose many challenges, both to traditional data warehousing architectures as well as to data mining algorithms and tools. Considerable research is being done on the algorithmic front, for example, using sampling or other data reduction techniques to cope with very large data sets, or through the development of scalable parallel algorithms for clustering, classification, association rules, and other data mining tasks. However, scaling the algorithms alone is insufficient. We have to think through the entire end−to−end architecture for delivering and deploying business intelligence applications, but relatively little work has been reported on such architectures.

Traditionally, data warehousing architectures were designed for relatively static reporting and analysis functions. Thus, the warehouse is loaded periodically (for example, nightly) by extracting, transforming, cleaning and consolidating data from several operational data sources that store transactional data (e.g., point−of−sale shopping transaction records in a retail application, call records in a telecommunication application). The data in the warehouse is then used to periodically generate reports, or to build multidimensional (data cube) views of the data for on−line querying and analysis, typically using on−line analytical processing (OLAP) tools. Increasingly, however, we are seeing business intelligence applications in industries such as telecommunications, electronic commerce, retail, and finance that are characterized by very high data volumes and data flow rates, and that require continuous analysis and mining of the data.

In this talk, we first summarize the requirements of business intelligence applications. Then, we describe an architectural approach that we are taking at HP Labs. to meet these requirements, and our experience with this approach. In this architecture, data flows continuously into a data warehouse, and is staged into one or more OLAP tools that are used as computation engines to continuously and incrementally build summary data cubes, which might then be stored back in the data warehouse. Analysis and data mining functions are performed continuously and incrementally over these summary cubes.

We discuss how the architecture attempts to satisfy a number of requirements of the applications. First, we address the problem of building the data cubes fast enough to match the input data flow rates. Next, we address the problem of retiring data from the warehouse. Retirement policies define when to discard data from the warehouse (i.e., move data from the warehouse into off−line archival storage). Data at different levels of aggregation may have different life spans depending on how they are to be used for downstream analysis and data mining. Then, we show how to use data cubes and OLAP tools to perform some data mining tasks such as customer profiling, computing similarity measures over customer profiles, and computing association rules. Finally, we show how to extend the architecture to do distributed warehousing and mining. In many applications, the data itself is distributed (e.g., a retail chain may wish to mine transaction data from different stores, call data may be collected from different monitoring points in a telecom. network). Distributing the data warehouse and the mining tasks can be more efficient and scalable than shipping all the data to a centralized warehouse and mining at a single site. However, distributing the mining task is not straightforward. For example, computing association rules over data from each store separately loses potentially information about customers who shopped at different stores. We illustrate an approach to distributed, cooperative data mining in which each local processing site produces intermediate summary data, which is forwarded to a global site for final processing.

To summarize, we describe an architecture for business intelligence applications that has the following key features: incremental data reduction using OLAP engines to generate summaries and enable data mining; staging of large volumes and flows of data having different life spans at different levels of aggregation; scheduling of operations on data depending on the type of processing to be performed and the age of the data; and distributed computation of summaries to enable large−scale mining.

For more information, the reader is referred to the following papers:

1. Q. Chen, U. Dayal, M.Hsu, "A Distributed OLAP Infrastructure for E−Commerce." *Proceedings, Fourth IFCIS International Conference on Cooperative Information Systems*, Sept. 1999.
2. Q.Chen, U. Dayal, M. Hsu, "A Data Warehousing and OLAP−Based Infrastructure for Business Intelligence Applications." *Proceedings, First International Conference on Data Warehousing and Knowledge Discovery*, Aug. 1999.

Parallel Branch-and-Bound Graph Search for Correlated Association Rules

Shinichi Morishita* and Akihiro Nakaya†
University of Tokyo

Abstract

There have been proposed efficient ways of enumerating all the association rules that are interesting with respect to support, confidence, or other measures. In contrast, we examine the optimization problem of computing the optimal association rule that maximizes the significance of the correlation between the assumption and the conclusion of the rule. We propose a parallel branch-and-bound graph search algorithm tailored to this problem. The key features of the design are (1) novel branch-and-bound heuristics, and (2) a rule of rewriting conjunctions that avoids maintaining the list of visited nodes. Experiments on two different types of large-scale shared-memory multi-processors confirm that the speed-up of the computation time scales almost linearly with the number of processors, and the size of search space could be dramatically reduced by the branch-and-bound heuristics.

1 Introduction

Many organizations are seeking strategies for processing or interpreting massive amounts of data that will inspire new marketing strategies or lead to the next generation of scientific discoveries. In response to those demands, in recent years, decision support systems and data mining systems have rapidly attracted strong interests, and numerous optimization techniques for computing decision trees, clusters, and association rules have been proposed. Among those techniques, the development of efficient ways of computing association rules has attracted considerable attention.

Association Rules

Given a set of records, an association rule is an expression of the form $X \Rightarrow Y$, where X and Y are tests on records, and X and Y are called the *assumption* and the *conclusion*, respectively. Consider the market basket analysis problem [1]. An example of an association rule is: "50% of customers who purchase bread also buy butter; 20% of customers purchase both bread and butter." We will describe the rule by

$$(Bread = 1) \Rightarrow (Butter = 1).$$

We call 50% the *confidence* of the rule and 20% the *support* of the rule.

The significance of an association rule has been evaluated by support and confidence [1, 2]. Higher support implies that the coverage of the rule is sufficiently large, while higher confidence shows that the prediction accuracy of using the assumption X as a test for inferring the conclusion Y is sufficient. In their pioneering work, Agrawal et al. [1, 2] define that an association rule is interesting if its support and confidence are no less than given thresholds, and they propose *Apriori* algorithm that enumerates all the interesting association rules. The idea of Apriori algorithm has been explored by many researchers [2, 8, 9, 10, 11, 12].

Motivating Example

Higher support and higher confidence, however, are not necessarily sufficient for evaluating the correlation between the assumption and the conclusion of an association rule. Brin et al. [5] address this problem, and the following example illustrates this issue.

Example 1.1 Consider the super market basket analysis problem [1]. Let $Bread, Butter$ and $Battery$ be Boolean attributes. Suppose that the support and the confidence of the following rule are 29% and 48.3%, respectively:

$$(Bread = 1) \wedge (Butter = 1) \Rightarrow (Battery = 1),$$

which means that customers who purchase both bread and butter may also buy batteries. This implication differs from our common sense, but the support and the confidence are fairly high, and hence one may conclude that the rule presents some unknown behavior of the customers. From a statistical viewpoint, however, we also ought to look at the negative implication that when customers who

*moris@ims.u-tokyo.ac.jp
†nakaya@ims.u-tokyo.ac.jp

	$(Battery = 1)$	$not(Battery = 1)$	Sum
$(Bread = 1) \wedge (Butter = 1)$	29	31	60
$not((Bread = 1) \wedge (Butter = 1))$	21	19	40
Sum	50	50	100

Table 1: Contingency Table

do not purchase both bread and butter may also buy batteries. In Table 1, which is called a contingency table, the row $(Bread = 1) \wedge (Butter = 1)$ and the row $not((Bread = 1) \wedge (Butter = 1))$ show the number of customers who do and do not meet $(Bread = 1) \wedge (Butter = 1)$, while the column $(Battery = 1)$ and the column $not(Battery = 1)$ shows their corresponding numbers, similarly.

Note that $Battery = 1$ holds for 50% of all the customers, which is higher than 48.3%, and hence customers satisfying $(Bread = 1) \wedge (Butter = 1)$ are less likely to meet $Battery = 1$. Thus there is a slight negative correlation between $(Bread = 1) \wedge (Butter = 1)$ and $Battery = 1$, though it is not significant. ∎

The above example suggests that we should measure the statistical significance of the correlation between the assumption and the conclusion. To measure the significance of correlation, the χ^2 value has usually been applied to the contingency table associated with the rule. The benefit of using the χ^2 value is that we can evaluate the significance of an association rule by a single value rather than multiple values such as support and confidence. All association rules can be ordered by their χ^2 values. We are then interested in finding the optimal association rule that maximizes the χ^2 value. Or we want to list the best n association rules in descending order of χ^2 value. We can also provide a cutoff value — say, at the 95% significance level — for χ^2, and then we can enumerate all the association rules whose χ^2 values are no less than that threshold. We will consider those problems, and we call an association rule *correlated* if its χ^2 value is optimal, sub-optimal or no less than a given threshold value.

Related Work

Brin et al.[5] have studied this problem from a slightly different aspect. Instead of finding correlated association rules, they focus on the computation of a set of primitive tests that are not independent by the chi-squared test. Using the strategy of Apriori algorithm [2], they present an algorithm for enumerating all the sets of primitive tests that are not independent, but the algorithm is not intended to compute correlated association rules.

Example 1.2 Let us consider the market basket analysis problem again. Suppose that $(Spaghetti = 1)$, $(Tabasco = 1)$, and $(Battery = 1)$ are not independent, because $(Spaghetti = 1)$ and $(Tabasco = 1)$ are correlated. We however cannot conclude that $(Spaghetti = 1) \wedge (Tabasco = 1) \Rightarrow (Battery = 1)$ is a correlated association rule, since the assumption and the conclusion may not be correlated at all. ∎

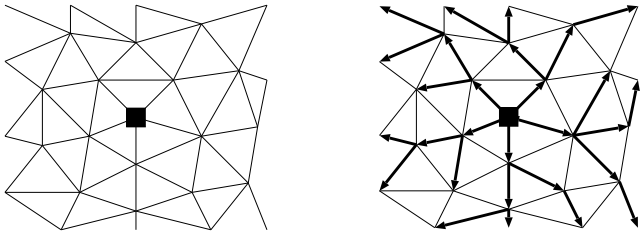
One may try to use Brin et al.'s algorithm to enumerate instances of $X \cup Y$ that are not independent and then try to derive correlated association rules. But there could be numerous instances of $X \cup Y$ from which no correlated association rules could be created, because even if primitive tests in X are correlated, X and Y are not correlated at all.

To keep the computation efficient, Brin et al. use a minimum support threshold as a pruning criteria. In practice, selecting a minimum support threshold requires some considerations, because using a higher threshold often results in pruning important patterns with lower support, while using a lower threshold might produce a huge amount of patterns, which is computationally costly. From the viewpoint of statistics, only the χ^2 value is essential, and hence Brin et al. discuss the possibility of avoiding the heuristics of using the minimum support threshold. We will work in this direction.

Overview

We define our problem more formally. Given a set of Boolean attributes, we select B as a special attribute and call it an *objective* attribute, while we call all the other attributes *conditional*. We use conditional attributes in the assumption of a rule. Consider all the association rules of the form "$(A_1 = v_1) \wedge \ldots \wedge (A_k = v_k) \Rightarrow (B = 1)$," where $v_i = 0$ or 1. We first remark that it is NP-hard to compute the optimal conjunction in the assumption that maximizes the χ^2 value. One may try to modify Apriori algorithm to compute the optimal conjunction, but this approach may not be promising, because Apriori algorithm is designed to enumerate all the possible association rules of interest, while our optimization problem targets the optimal conjunction or sub-optimal ones.

To cope with such optimization problems, one common approach is an iterative improvement



(a) (b)

Figure 1: The figure (a) shows the search space of conjunctions. The figure (b) shows the distributed search tree rooted at the square black dot.

graph search algorithm that initially selects a candidate conjunction by using a greedy algorithm and then tries to improve the ensemble of candidate conjunctions by a local search heuristic; that is, from a conjunction we generate a *neighboring* conjunction that is obtained by replacing one primitive test with another, by deleting a test, or by inserting a new test. Figure 1(a) represents the search space of all conjunctions by an undirected graph in which a pair of neighboring conjunctions is connected by an edge. Starting from the initial conjunction represented by the square dot, we want to search the graph without visiting the same node more than once. Figure 1(b) illustrates such an example.

To accelerate the performance of graph search, parallelizing the search has been studied for various discrete optimization problems [3, 6]. We will exploit this approach for searching the optimal conjunction. To avoid the repetition of visiting the same node, conventional graph search algorithms maintain the list of visited nodes [3, 6], which however could be a severe bottleneck of parallel search. We instead propose a rule of rewriting a conjunction to others. We first apply the rewriting rule to the initial conjunction to obtain child conjunctions, and then we repeat application of the rule to descendant conjunctions so that we can visit every conjunction just once without maintaining the list of visited conjunctions. Moreover, each application of the rewriting rule can be well parallelized.

If the initial conjunction is empty, it is rather trivial to build such a search tree. For instance, we can create one child of a conjunction by inserting one primitive test. In general, however, an arbitrary conjunction could be selected as the initial conjunction, and we need to create a neighboring conjunction by using one of replacement, deletion, or insertion, which makes the extraction of a search tree non-trivial.

To reduce the size of the search tree, we develop a branch-and-bound heuristics appropriate for the significance of correlation. We also develop implementation techniques such as materialization of projections and maintenance of distributed priority queues.

2 Preliminaries

Attributes, Records, and Primitive Tests

The domain of a Boolean attribute is $\{0, 1\}$, where 0 and 1 represent true and false, respectively. Let B be a Boolean attribute, let t denote a record (tuple), and let $t[B]$ be the value for attribute B. A *primitive* test has the form $B = v$ where v is either 0 or 1. A record t meets $B = v$ if $t[B] = v$. A conjunction of primitive tests $t_1, t_2, \ldots, t_k$ is of the form $t_1 \wedge t_2 \wedge \ldots \wedge t_k$. A record t meets a conjunction of primitive tests if t satisfies all the primitive tests. We simply call primitive tests and conjunctions *tests*.

Association Rules

From a given set of Boolean attributes, we select one as a special attribute and call it the *objective* attribute. We call all the other attributes *conditional*. Let B be the objective attribute. An *association rule* has the form:

$$(A_1 = v_1) \wedge \ldots \wedge (A_k = v_k) \Rightarrow (B = v),$$

where $A_i (i = 1, \ldots, k)$ is an conditional attribute, and each of v_i and v is either 0 or 1. For instance, $(Bread = 1) \wedge (Butter = 1) \Rightarrow (Battery = 1)$ is an association rule.

Consider association rule $X \Rightarrow Y$. Let R be a set of records over $\mathcal{R}$, and let $|R|$ denote the number of records in R. Let R_1 be the set of records that meet the assumption X, and let R_2 denote $R - R_1$. We call a record t *positive* (*negative*, resp.) if t satisfies (does not meet) the conclusion Y. Let R^t and R^f denote the set of positive and negative records in R, respectively. Table 2 summarizes numbers of records that meet each condition. Since R is given and fixed, we assume that $|R|$, $|R^t|$, and $|R^f|$ are constants, but $|R_1^t|$, $|R_2^t|$, $|R_2^f|$, and $|R_2^f|$ may vary according to the choice of the assumption X. Let n and m denote $|R|$ and $|R^t|$ respectively, then

	Y is true.	Y is false	Sum of Row
X is true.	$\|R_1^t\|(=y)$	$\|R_1^f\|(=x-y)$	$\|R_1\|(=x)$
X is false.	$\|R_2^t\|(=m-y)$	$\|R_2^f\|(=n-x-(m-y))$	$\|R_2\|(=n-x)$
Sum of Column	$\|R^t\|(=m)$	$\|R^f\|(=n-m)$	$\|R\|(=n)$

Table 2: Contingency Table

$|R^f| = n - m$. Let x and y denote $|R_1|$ and $|R_1^t|$ respectively. Observe that if we specify the values of x and y, the values of all the other variables are determined.

Chi-squared Value

The chi-squared value is a normalized deviation of observation from expectation. Table 2 presents observed numbers of records. Expected numbers are calculated as follows: In the entire relation, the probability that a positive record occurs is $\frac{|R^t|}{|R|} = \frac{m}{n}$. Since the observed number of records satisfying X is $|R_1|$, the expected number of records meeting both X and Y is $|R_1|$ times $\frac{m}{n}$. Table 3 presents expected numbers of records. The chi-

	Y is true.	Y is false
X is true.	$\|R_1\|\frac{m}{n}$	$\|R_1\|\frac{n-m}{n}$
X is false.	$\|R_2\|\frac{m}{n}$	$\|R_2\|\frac{n-m}{n}$

Table 3: Expected Numbers of Records

squared value is defined as the total of the squared difference between the observed number and the expected number divided by the expected number for each cell; that is,

$$\frac{(|R_1^t| - |R_1|\frac{m}{n})^2}{|R_1|\frac{m}{n}} + \frac{(|R_1^f| - |R_1|\frac{n-m}{n})^2}{|R_1|\frac{n-m}{n}} + \frac{(|R_2^t| - |R_2|\frac{m}{n})^2}{|R_2|\frac{m}{n}} + \frac{(|R_2^f| - |R_2|\frac{n-m}{n})^2}{|R_2|\frac{n-m}{n}}.$$

Since all the variables are determined by x and y, we will refer the above formula by $\chi^2(x, y)$. If X and Y are independent, the observed number is equal to the expected number (in this case, $\frac{y}{x} = \frac{m}{n}$), and therefore $\chi^2(x, y)$ is equal to 0. In the chi-squared test, if $\chi^2(x, y)$ is greater than a cutoff value – say, at the 95% significance level —, we reject the independence assumption.

Convexity of Function

Let $\phi(x, y)$ be a function that is defined on $(x, y) \in D$. $\phi(x, y)$ is a *convex* function on D if for any (x_1, y_1) and (x_2, y_2) in D and any $0 \leq \lambda \leq 1$,

$$\begin{aligned} & \phi(\lambda(x_1, y_1) + (1 - \lambda)(x_2, y_2)) \\ \leq \; & \lambda\phi(x_1, y_1) + (1 - \lambda)\phi(x_2, y_2). \end{aligned}$$

Let $(x_3, y_3) = \lambda(x_1, y_1) + (1 - \lambda)(x_2, y_2)$, then $\phi(x_3, y_3) \leq \max(\phi(x_1, y_1), \phi(x_2, y_2))$.

Proposition 2.1 $\chi^2(x, y)$ is a convex function defined on $0 \leq y \leq x$.

Proof: (Sketch) For any δ_1 and δ_2, define $V = \delta_1 x + \delta_2 y$. Prove $\partial^2\chi^2(x, y)/\partial V^2 \geq 0$. ∎

The convexity of $\chi^2(x, y)$ is crucial to prove the intractability of computing the optimal conjunction. We also use the convexity to derive an effective branch-and-bound heuristics.

Theorem 2.1 Let S denote a set of conjunctions that use conditional attributes, and Y be the objective conclusion. It is NP-hard to find the optimal conjunction $X \in S$ such that the chi-squared value of $X \Rightarrow Y$ is maximum.

Proof: (Sketch) The case for the entropy value is proved in [7]. In the proof, the convexity of the entropy function is essentially used. The argument carries over to the case for the chi-squared value, because the chi-squared function is also convex. ∎

3 Parallel Branch-and-Bound Graph Search

Search Space as an Undirected Graph

Let V denote the set of all conjunctions that use conditional attributes. A conjunction C_1 is *adjacent to* another conjunction C_2 if C_1 is obtained by replacing a primitive test in C_2 with another, by deleting a primitive test in C_2, or inserting a new one to C_2.

Example 3.1 Let C be the conjunction $(A_1 = 1) \wedge (A_2 = 0) \wedge (A_3 = 1)$. C is adjacent to $(A_1 = 1) \wedge (A_2 = 0) \wedge (A_4 = 0)$, because $(A_3 = 1)$ in C is replaced by $(A_4 = 0)$. Also, C is adjacent to $(A_1 = 1) \wedge (A_3 = 1)$ and $(A_1 = 1) \wedge (A_2 = 0) \wedge (A_3 = 1) \wedge (A_5 = 1)$. ∎

Let E denote the set of undirected edges between pairs of adjacent nodes in V; that is, $E = \{(C_1, C_2) \mid C_1 \text{ is adjacent to } C_2\}$. The undirected graph (V, E) represents the search space of all conjunctions. We call (V, E) the *undirected graph* of

adjacency. Figure 1(a) in Section 1 presents an example. We define the *distance* between nodes v and u by the length of the shortest path between v and u. Put another way, the distance shows the minimum number of operations on primitive tests to generate u from v.

Requirements on Search Tree

Suppose that we are given an arbitrary node $t_1 \wedge \ldots \wedge t_k$ in the search space (V, E) as the initial conjunction. To realize the local search strategy starting from $t_1 \wedge \ldots \wedge t_k$, we need to generate a search tree rooted at $t_1 \wedge \ldots \wedge t_k$ such that (1) the depth from $t_1 \wedge \ldots \wedge t_k$ to any node v in the tree is equal to the distance between $t_1 \wedge \ldots \wedge t_k$ and v in (V, E), and (2) each conjunction is enumerated to appear just once in the tree. For instance, Figure 1(b) illustrates such a search tree rooted at the square black dot.

To build a search tree, we first present a way of creating a unique path from the root of the initial conjunction to the node of any conjunction. We then show how to assemble all the paths into a search tree.

Creating a Unique Path from the Initial Conjunction to Any Conjunction

We introduce a way of representing a conjunction uniquely with respect to the initial conjunction. We develop this idea motivated by techniques for enumerating geometric objects [4]. We assume that all the primitive tests are sorted in a total order, and we denote the order by $x_1 < x_2$. For simplicity of presentation, we introduce a dummy test $\perp$ that is strictly smaller than any test t; that is, $\perp < t$. Let S denote the set of all the primitive tests. Let $t_1 \wedge \ldots \wedge t_k$ be the the initial conjunction given. Let C denote an arbitrary conjunction of primitive tests in S. We represent C by a list of primitive tests according to the following steps:

1. If $t_i (1 \leq i \leq k)$ appears in C, place t_i at the i-th position in the list. Otherwise, leave the i-th position open.
2. Sort all the primitive tests that appear in C but are not in $\{t_1, \ldots, t_k\}$, in the ascending order. Let SL denote the sorted list. Select and remove the first primitive test in SL, and assign it to the leftmost open position. Repeat this process until SL becomes to be empty.

Observe that any conjunction can be represented by the unique list of primitive tests, and hence we call it the *canonical* list.

Example 3.2 Let $t_1 \wedge t_2 \wedge t_3 \wedge t_4 \wedge t_5$ be the initial conjunction. Its canonical list is $[t_1, t_2, t_3, t_4, t_5]$. Let $\bigcirc$ denote an open position. The canonical list of $t_4 \wedge a_1 \wedge t_2 \wedge a_2 \wedge t_5 \wedge a_3$, where $a_1 < a_2 < a_3$, is obtained as follows: We first create $[\bigcirc, t_2, \bigcirc, t_4, t_5]$ by placing each t_i of $t_4 \wedge a_1 \wedge t_2 \wedge a_2 \wedge t_5 \wedge a_3$ at the i-th position. We then assign a_1 and a_2 respectively to the first and the third positions, which are open, and we append a_3 at the end of the list. Consequently we have $[a_1, t_2, a_2, t_4, t_5, a_3]$.

The canonical list of $t_4 \wedge a_1 \wedge a_2$, where $a_1 < a_2$, is obtained similarly. We first create $[\bigcirc, \bigcirc, \bigcirc, t_4, \bigcirc]$, and then assign a_1 and a_2 into the first and the second positions, respectively. Thus we obtain $[a_1, a_2, \bigcirc, t_4, \bigcirc]$. ∎

We show how to rewrite the canonical list of the initial conjunction to that of an arbitrary target conjunction, which creates a unique path from the root to any node. Intuitively, we scan two lists together from left to right, and when we find different primitive tests at the same position, we perform one of replacement, deletion, or insertion so that the initial conjunction is transformed into the target conjunction after the scan.

Example 3.3 Consider canonical lists $[t_1, t_2, t_3, t_4, t_5]$ and $[a_1, t_2, a_2, t_4, t_5, a_3]$. Since the two primitive tests at the first position are different, we replace t_1 by a_1. We then see the difference at the third position, and we replace t_3 by a_2. Finally, a_3 at the sixth position of $[a_1, t_2, a_2, t_4, t_5, a_3]$ does not appear in $[t_1, t_2, t_3, t_4, t_5]$, and hence we insert a_3. Consequently we have rewritten $[t_1, t_2, t_3, t_4, t_5]$ to $[a_1, t_2, a_2, t_4, t_5, a_3]$ by the following sequence of operations:

$$[t_1, t_2, t_3, t_4, t_5] \overset{\text{replacement}}{\rightarrow} [a_1, t_2, t_3, t_4, t_5] \overset{\text{replacement}}{\rightarrow} [a_1, t_2, a_2, t_4, t_5] \overset{\text{insertion}}{\rightarrow} [a_1, t_2, a_2, t_4, t_5, a_3]$$

We have applied three operations, and the distance between $t_1 \wedge t_2 \wedge t_3 \wedge t_4 \wedge t_5$ and $a_1 \wedge t_2 \wedge a_2 \wedge t_4 \wedge t_5 \wedge a_3$ in the graph of conjunctions is also 3. ∎

There are some issues on sequences that use deletion.

Example 3.4 Consider the following two sequences

- $[t_1, t_2, t_3, t_4, t_5] \overset{\text{deletion}}{\rightarrow} [\bigcirc, t_2, t_3, t_4, t_5] \overset{\text{insertion}}{\rightarrow} [a_1, t_2, t_3, t_4, t_5]$
- $[t_1, t_2, t_3, t_4, t_5] \overset{\text{replacement}}{\rightarrow} [a_1, t_2, t_3, t_4, t_5]$

The second sequence gives the minimum length path in the undirected graph of adjacency. The following two sequences show another issue:

- $[a_1, t_2, t_3, t_4, t_5, t_6] \overset{\text{deletion}}{\rightarrow} [a_1, \bigcirc, t_3, t_4, t_5, t_6] \overset{\text{replacement}}{\rightarrow} [a_1, \bigcirc, a_2, t_4, t_5, t_6]$

- $[a_1, t_2, t_3, t_4, t_5, t_6] \overset{\text{replacement}}{\rightarrow} [a_1, a_2, t_3, t_4, t_5, t_6] \overset{\text{deletion}}{\rightarrow} [a_1, t_2, \bigcirc, t_4, t_5, t_6]$

$[a_1, \bigcirc, a_2, t_4, t_5, t_6]$ is not a canonical list, because $\bigcirc$ appears before a_2. ∎

In each case of the above example, we want to derive the second sequence only. We can solve this problem by using the rule that we do not allow replacement nor insertion once deletion is used. We will prove that this restriction does not overlook the canonical list of any conjunction.

Making Canonical Lists Distributable to Arbitrary Multiple Processes

We present a way of distributing the canonical lists of conjunctions to arbitrary multiple processes so that each process can continue to rewrite independently. Consider the following sequence again:

$[t_1, t_2, t_3, t_4, t_5] \overset{\text{replacement}}{\rightarrow} [a_1, t_2, t_3, t_4, t_5] \overset{\text{replacement}}{\rightarrow} [a_1, t_2, a_2, t_4, t_5] \overset{\text{insertion}}{\rightarrow} [a_1, t_2, a_2, t_4, t_5, a_3]$

Suppose that we assign the third canonical list $[a_1, t_2, a_3, t_4, t_5]$ to one process. We want to avoid giving to the process the history of creating the previous two canonical lists, because in general the history could be lengthy. We rather provide minimum information to the process so that the process can continue to rewrite the canonical list. For instance, it is enough to provide the information that primitive tests up to the third position have been updated, a_2 is the largest primitive test that has been most recently added, and no primitive test has been deleted. With this information, we can then append a_3, which is greater than a_2, at the end of $[a_1, t_2, a_2, t_4, t_5]$.

In general, we add the following auxiliary information to a canonical list $[x_1, \ldots, x_n]$, and we represent the extension by $\langle [x_1, \ldots, x_n], n, i, max, dmode \rangle$, which we also call a *canonical* list.

- n: The number of primitive tests in the canonical list.
- i: Let j be an index such that $i \leq j$. We can update the primitive test at the j-th position in the next step.
- max: max denotes the largest primitive test among all the primitive tests that have been added. In the next step, we need to add a new primitive test that is greater than max when we perform replacement or insertion. For the initial conjunction we set max to $\perp$, where $\perp$ is the dummy test smaller than any primitive test.
- $dmode$: For the initial conjunction, $dmode = 0$. Once deletion is applied, $dmode$ is set to 1. When $dmode = 1$, only deletion is applicable.

Application of replacement, insertion or deletion to is defined as follows:

- **Replacement**: When $i \leq n$ and $dmode = 0$, we can replace the j-th ($j \geq i$) primitive test with x such that $max < x$ and $x \notin \{t_1, \ldots, t_k\}$.

  $\langle [x_1, \ldots, x_n], n, i, max, 0 \rangle \overset{\text{replacement}}{\rightarrow} \langle [x_1, \ldots, x_{j-1}, x, x_{j+1}, \ldots, x_n], n, j+1, x, 0 \rangle$.

- **Insertion**: When $dmode = 0$, we can insert x such that $max < x$ and $x \notin \{t_1, \ldots, t_k\}$ at the end of the list.

  $\langle [x_1, \ldots, x_n], n, i, max, 0 \rangle \overset{\text{insertion}}{\rightarrow} \langle [x_1, \ldots, x_n, x], n+1, n+2, x, 0 \rangle$.

- **Deletion**: When $i \leq n$, we can delete the j-th ($j \geq i$) primitive test, and we set $dmode$ to 1.

  $\langle [x_1, \ldots, x_n], n, i, max, dmode \rangle \overset{\text{deletion}}{\rightarrow} \langle [x_1, \ldots, x_{j-1}, \bigcirc, x_{j+1}, \ldots, x_n], n-1, j+1, max, 1 \rangle$

Table 4 presents two examples of such sequences.

Theorem 3.1 Let A and B denote a given initial conjunction and an arbitrary conjunction. There exists a unique sequence of application of replacement, insertion or deletion that rewrites the canonical list of A to that of B. Furthermore, the number of instances of application is equal to the distance between A and B in the undirected graph of adjacency. ∎

Proof: (Sketch) The proof is an induction on the number of positions where the two primitive tests do not coincide in A and B. ∎

Distributable Search Tree

The *distributable search tree* is a binary tree that displays all the sequences from the initial canonical list to the canonical list of any conjunction. Figure 2 illustrates such an example. Theorem 3.1 implies that any distributable search tree meets the two requirements on search trees; that is, (1) the depth from the root to any node v in the tree is equal to the distance between the root and v in the graph of adjacency, and (2) each conjunction is enumerated to appear just once in the tree.

Furthermore any node in a distributable search tree can be assigned to any process in a flexible manner.

	$\langle [t_1, t_2, t_3, t_4, t_5], 5, 1, \perp, 0\rangle$		$\langle [t_1, t_2, t_3, t_4, t_5, t_6], 6, 1, \perp, 0\rangle$
replacement $\rightarrow$	$\langle [a_1, t_2, t_3, t_4, t_5], 5, 2, a_1, 0\rangle$	replacement $\rightarrow$	$\langle [a_1, t_2, t_3, t_4, t_5, t_6], 6, 2, a_1, 0\rangle$
replacement $\rightarrow$	$\langle [a_1, t_2, a_2, t_4, t_5], 5, 4, a_2, 0\rangle$	replacement $\rightarrow$	$\langle [a_1, a_2, t_3, t_4, t_5, t_6], 6, 3, a_2, 0\rangle$
insertion $\rightarrow$	$\langle [a_1, t_2, a_2, t_4, t_5, a_3], 6, 7, a_3, 0\rangle$	deletion $\rightarrow$	$\langle [a_1, a_2, \bigcirc, t_4, t_5, t_6], 5, 4, a_2, 1\rangle$
		deletion $\rightarrow$	$\langle [a_1, a_2, \bigcirc, t_4, \bigcirc, t_6], 4, 6, a_2, 1\rangle$

Table 4: Examples of Distributable Sequences

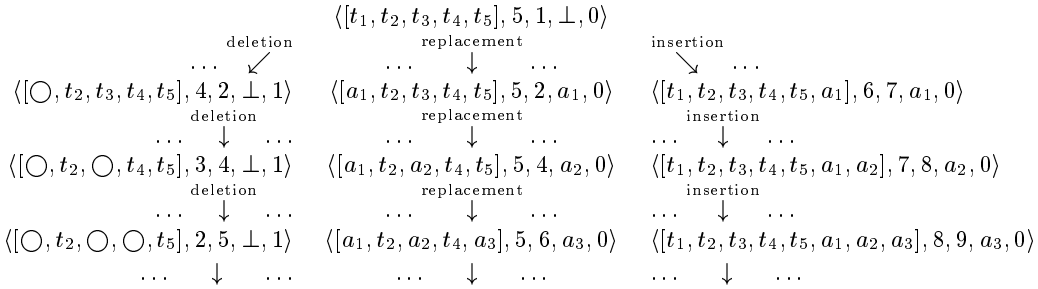


Figure 2: Example of Distributable Search Tree

Best-First Search

Next we discuss how to traverse the distributed search tree. Suppose that we compute an initial conjunction by a greedy algorithm that always makes the choice that looks best at the moment. Next we need to consider how to scan the distributed search tree rooted at the initial conjunction. One may try the depth-first search or the breadth-first search, but when we look for the conjunction that maximizes the chi-squared value, we should make a locally optimal choice in hope that this choice will lead to the global optimal solution. We therefore select the best-first search strategy that expands a node whose chi-squared value is maximum at the moment.

We implement the best-first search by using a priority queue. First we insert the initial conjunction into the empty queue. We repeat the process that we remove the first node v from the queue, compute the chi-squared value of v, update the best chi-squared value if necessary, use the chi-squared value of v to prioritize each child of v, and insert all the chide nodes of v into the queue.

Later in this section we show how to distribute the queue to multiple processes, but before that we show two techniques to improve the performance of the best-first search.

Branch-and-Bound Heuristics

Suppose that we examine a node v in a distributable search tree. The following theorem shows how to compute an upper bound of the best chi-squared value that could be obtained by scanning all the nodes in the subtree rooted at v. If the upper bound is smaller than the optimum chi-squared value at the moment, we can ignore and prune the subtree.

Theorem 3.2 Let v be a node in a distributed tree. Suppose that

$$v = \langle [x_1, \ldots, x_i, x_{i+1}, \ldots, x_k], k, i+1, -, -\rangle.$$

Let a (b, resp.) denote the number of (positive) records that meet $x_1 \wedge \ldots \wedge x_i$. Let w be an arbitrary descendant of v. Note that the conjunction of w contains $x_1 \wedge \ldots \wedge x_i$. Let p (q, resp.) denote the number of (positive) records that meet the conjunction of w. Recall that $\chi^2(p, q)$ is the chi-squared value of w, and we have:

$$\chi^2(p, q) \;\leq\; \max\{\chi^2(b, b), \chi^2(a - b, 0)\}.$$

Proof: (Sketch) Let n and m denote respectively the number of records and the number of positive records in the entire relation. Consider the points (a, b) and (p, q) in the two-dimensional Euclidean plane. It is easy to see that (p, q) falls in the convex region whose vertexes are $(0, 0)$, (b, b), (a, b), and $(a - b, 0)$. To be more precise, we have $0 \leq p \leq a \leq n$, $0 \leq q \leq b \leq m$, $q \leq p$, $b \leq a$, and $(p - q) \leq (a - b)$. When $y/x = m/n$, $\chi^2(x, y)$ is zero and minimum. Because of the convexity of $\chi^2(x, y)$, it follows that $\chi^2(p, q) \leq \max\{\chi^2(b, b), \chi^2(a - b, 0)\}$. ∎

Materialized Projections

Let $v = \langle [x_1, \ldots, x_i, x_{i+1}, \ldots, x_{n_1}], n_1, i+1, -, -\rangle$ be an arbitrary node in a distributed tree. Note

that any node in the subtree rooted at v must contain all the primitive tests in $\{x_1, \ldots, x_i\}$, because none of $\{x_1, \ldots, x_i\}$ is updated in the subtree. We call the set of records that meet $x_1 \wedge \ldots \wedge x_i$ the *materialized projection* for v.

A materilized projection could be very large in practice. To utilize the main memory efficiently, we implemented a materialized projection by creating a bit array of indexes to records in the materialized projection. If the bit of an index is on, the record of the corresponding index belongs to the materialized projection. For instance, the size of a bit array for a large database containing ten million records is $1.25MB$. Such bit arrays may still require large memory space during the execution, especially when the queue becomes to be long during the computation and cannot fit in the main memory. In this case, we put aside nodes with lower priorities, which might not be processed for a while, to the secondary disk at the moment, and later we restore them back to the main memory.

We now discuss the benefit of associating the materialized projection with each node. Let $w = \langle [x_1, \ldots, x_i, y_{i+1}, \ldots, y_{n_2}], n_2, j+1, -, - \rangle$ be a descendant of v. When we compute the chi-squared value of w, we need to count the number of records that satisfy the conjunction of w. It suffices to check if each record in the materialized projection for v also satisfies $y_{i+1} \wedge \ldots \wedge y_{n_2}$. The materialized projection could be much smaller than the entire relation. Since counting the number of records that satisfy a conjunction is the crucial step of the whole computation, the use of materialized projections could reduce the computation time substantially. The materialized projection of each node can be computed in an incremental manner; that is, the materialized projection for a child node is a subset of that for its parent.

Distributing Priority Queue to Multiple Processes

It remains to parallelize the single process version of the best-first search. The key extension is to divide the single queue into multiple disjoint queues and to distribute them to multiple processes. Balancing sizes of queues among multiple processes at run time is rather straightforward, because any node can be processed by any process. Each process maintains its own queue and broadcasts the locally best chi-squared value to the others when the value is updated.

There are a couple of concerns that do not arise previously. The first issue is that broadcasting the update of the locally best chi-squared value may increase the communication overhead between the processes. Another concern is that short delay of the broadcast may slightly deteriorate the overall performance, because the branch-and-bound heuristics uses the best chi-squared value at the moment. Tests however show that updates do not occur so often, and therefore those concerns are not serious in practice.

Listing Best n Conjunctions

We have so far presented an algorithm for computing the optimal conjunction, but it is easy to modify the algorithm to list the best n conjunctions. To this end, we can change to maintain the list of the best n conjunctions instead of the best conjunction. After this modification, the branch-and-bound heuristics still works, because we can use the n-th node instead of the best node to prune the search space according to Theorem 3.2.

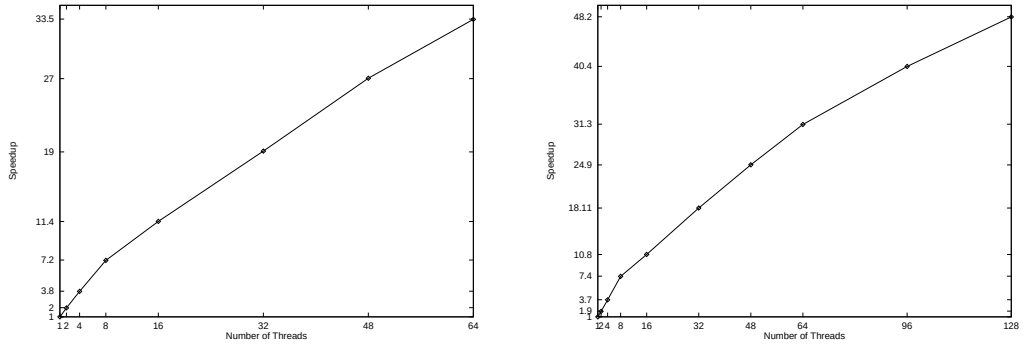
4 Experimental Results

Implementation

We implemented our algorithm by using C++ and POSIX thread library. Experiments were performed on two different types of large scaled shared-memory multi-processors. One is Sun Microsystems Ultra Enterprise 10000 with 64 UltraSPARC processors running at 250MHz, 16GB of main memory, and 1MB of L2 cache for each processor, working under Solaris 2.5.1. Another is SGI Origin 2000 with 128 R10000 processors running at 195MHz, 24GB main memory, and 4MB L2 cache for each processor, running under IRIX 6.5SE. We limit the size of main memory to 2GB in order to verify that our implementation uses at most 2GB of main memory. In the case of SGI Origin 2000, since the time to access the remote memory is almost three times larger than the time to access the local memory, we had to implement each thread to keep a local copy of the entire relation to accelerate the overall performance.

Test Data

We randomly generated such a relation that the relation contains one hundred thousand records and the value of an attribute in a record is equal to 1 with a probability of p. We show the experimental results when $p = 0.3$, because in this case, the execution time was at most several hours, and therefore we can measure the speed-up and the effect of the branch-and-bound heuristics in a reasonable amount of time. The relation contains one hundred conditional attributes and one objective attribute. We used one hundred primitive tests of the form $(A = 1)$, where A is a conditional attribute. As the conclusion, we used $(B = 1)$, where B is the objective attribute. We selected the initial conjunction



(a) Sun Microsystems Ultra Enterprise 10000 (b) SGI Origin 2000

Figure 3: Speed-up Ratio

#(threads)	Enterprise 10000		Origin 2000	
	Execution Time	Speed-up Ratio	Execution Time	Speed-up Ratio
1	6,760	1.00	6,503	1.00
64	202	33.47	219	31.30
128	N/A	N/A	135	48.17

Table 5: Execution Time in Seconds and Speed-up Ratio

in a greedy manner. We applied our implementation to the test data until the algorithm terminates; that is, all the queues become to be empty, and the optimal conjunction is identified.

Effect of Branch-and-Bound Heuristics

Since there are one hundred primitive tests, the algorithm could generate 2^{100} conjunctions in the worst case. As a result of the branch-and-bound heuristics, however, the algorithm generates much less conjunctions. We have performed the cases when numbers of threads are 1, 2, 4, 8, 16, 32, 64, and 128. The total number of conjunctions inserted into the distributed queues ranges from 24194 to 24463. We have observed that every conjunction examined during the search contains at most four primitive tests. Note that the number of all conjunctions with at most four primitive tests is about 4.3×10^6. This figure again indicates that the branch-and-bound heuristics can drastically reduce the search space.

Speed-up

The *speed-up ratio* of n threads is defined as the ratio of the execution time of one thread to the execution time of n threads. Figure 3 (a) and (b) present that the speed-up scales almost linearly with the number of threads on both Sun Microsystems Ultra Enterprise 10000 and SGI Origin 2000. Table 5 shows the execution time in seconds.

Optimizing the Objective Criteria

Until the system finds the optimal conjunction, it outputs the optimal conjunction at the moment. Let X and Y denote the assumption and the conclusion of an association rule. Table 6 presents candidates of the optimal conjunction that the system output during the computation when the system was executed as 64 threads on Sun Enterprise 10000.

Relationship between Execution Time and Size of Search Space

We have so far presented the performance of our system applied to the set of one hundred thousand records such that the value of each attribute in a record is equal to 1 with a probability of $p = 0.3$. If we use higher values for the probability p, the number of conjunctions examined increases, and therefore the total execution time also grows. Table 7 summarizes the performance results of executing our algorithm as 32 threads on SUN Ultra Enterprise 10000. The execution time does not always scale to the number of conjunctions examined, since time to handle a longer conjunction with more primitive tests decreases because of the effect of using materialized projections. The execution time also depends on the structure of the search tree. But in general, the growth of the number of conjunctions raises the execution time.

Assumption X	X is true.		X is false.		χ^2
	Y is true.	Y is false.	Y is true.	Y is false.	
$[t_1, t_2]$	9807	4240	20575	65378	12014.836
$[a_{10}, t_2]$	5962	411	24420	69207	12841.383
$[a_{15}, a_{23}, a_{90}]$	5810	161	24572	69457	13445.606
$[t_1, a_{39}, a_{90}]$	5833	148	24549	69470	13559.018

Table 6: Candidates of Optimal Conjunctions Calculated During the Computation ($[t_1, t_2]$ is the initial conjunction, and $[t_1, a_{39}, a_{90}]$ is the optimal one.)

p	Execution Time in Seconds	Number of Conjunctions
0.3	354	24,463
0.4	1,396	74,169
0.5	2,525	233,148
0.6	8,261	803,280

Table 7: Relationship between the Performance and the Size of Search Space

5 Conclusion

We have examined the optimization problem of computing the optimal conjunction maximizing the chi-squared value that indicates the significance of the correlation between the assumption and the conclusion of the rule. Although this optimization problem is NP-hard, we have introduced a novel data structure called the distributable search tree, and we have presented how to construct this tree and how to speed up the performance of searching the distributable search tree on multiple processes. Our technique carries over to the general cases when we use the entropy function, the gini index, or the correlation coefficient as evaluation criteria.

In Section 4, we use a synthesis data to evaluate the performance of our system. In practice, we have been applying our system to the analysis of multiple factors leading to a common disease such as diabetes, or high blood sugar level. This case poses another problem of finding a conjunction to split data into two classes so that the average of the objective numeric attribute values in one class is substantially higher than that in the other class. It is however NP-hard to find the optimal conjunction that maximizes the interclass variance [7]. Developing an effective branch-and-bound heuristics for this case is an interesting problem.

Acknowledgements

All experimental results are done by using parallel machines at Human Genome Center, Institute of Medical Science, University of Tokyo. This research is partly supported by Grant-in-Aid for Scientific Research on Priority Areas "Discovery Science" from the Ministry of Education, Science and Culture, Japan.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of ACM SIGMOD*, pages 207–216, May 1993.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of VLDB Conference*, pages 487–499, 1994.

[3] G. Y. Ananth, V. Kumar, and P. Pardalos. Parallel processing of discrete optimization problems. 1993.

[4] D. Avis and K. Fukuda. A basis enumeration algorithm for linear systems with geometric applications. *Applied Mathematics Letters*, 5:39–42, 1991.

[5] S. Brin, R. Motwani, and C. Silverstein. Beyond market baskets: Generalizing association rules to correlations. In *Proceedings of ACM SIGMOD*, pages 265–276. *SIGMOD Record* 26(2), June 1997.

[6] V. Kumar, A. Grama, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin Cummings, Nov. 1993.

[7] S. Morishita. On classification and regression. In *Proceedings of Discovery Science, DS'98, Lecture Notes in Artificial Intelligence*, volume 1532, pages 40–57, Dec. 1998.

[8] R. T. Ng, L. V. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained association rules. In *Proceedings of ACM SIGMOD*, pages 13–24, June 1998.

[9] J. S. Park, M.-S. Chen, and P. S. Yu. An effective hash-based algorithm for mining association rules. In *Proceedings of ACM SIGMOD*, pages 175–186, May 1995.

[10] R. J. Bayardo Jr. Efficiently Mining Long Patterns from Databases. In *Proceedings of ACM SIGMOD*, pages 85–93, June 1998.

[11] R. J. Bayardo Jr., R. Agrawal, D. Gunopulos. Constraint-Based Rule Mining in Large, Dense Databases. In *Proceedings of ICDE*, pages 188-197, March 1999.

[12] R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. In *Proceedings of ACM SIGMOD*, June 1996.

Parallel Generalized Association rule Mining on Large Scale PC Cluster

Takahiko SHINTANI and Masaru KITSUREGAWA
Institute of Industrial Science, The University of Tokyo
7-22-1 Roppongi, Minato-ku, Tokyo 106, Japan
E-mail : {*shintani, kitsure*}@tkl.iis.u-tokyo.ac.jp

Abstract

One of the most important problems in data mining is discovery of association rules in large database. In our previous study, we proposed parallel algorithms and candidate duplication based load balancing strategies for mining generalized association rules and showed our algorithms could attain good performance in 16 nodes system. However, as the number of processors increase, it would be difficult to achieve flat workload distribution.

In this paper, we present the candidate partition based load balancing strategy for parallel algorithm of generalized association rule mining. This strategy partitions the candidate itemsets so that the number of candidate probes for each node is equalized each other with estimated support count by the information of previous pass. Moreover, we implement the parallel algorithms and load balancing strategies for mining generalized association rules on a cluster of 100 PCs interconnected with an ATM network, and analyze the performance of our algorithms using a large amount of transaction dataset. Through the several experiments, we showed the load balancing strategy which partition the candidate itemsets with considering the distribution of candidate probes and duplicate the frequently occurring candidate itemsets, can attain high performance and achieve good workload distribution on one hundred PC cluster system.

1 Introduction

Recently, PC (Personal computer) clusters have become a hot research topic in the field of parallel and distributed computing. Today's parallel computer systems are moving away from proprietary hardware components to commodity parts for CPUs, disks and memories. While an interconnection network has not yet been commoditized, ATM technology becomes the standard for high speed communication. Moreover, PC performance is increasing incredibly rapidly these days and the price of PCs remains inexpensive compared with that of workstation. Thus looking over recent technology trends, ATM connected PC clusters are very promising platform for massively parallel processing. We developed a PC cluster system consisting of 100 PCs for parallel processing. There have been performed various research projects to develop PC clusters[Tamura *et al.*, 1997]. We believe that data intensive applications such as data mining are very important applications for parallel processing in addition to the conventional scientific applications.

Data mining has attracted a lot of attention for discovering useful information such as rules and previously unknown patterns existing between data items embedded in large databases, which allows effective utilization of large amount of accumulated transaction log. Association rule mining is one of the most important problems in data mining. Association rule is the rule about what items are bought together within the transaction, such as "70% of the customers who buy A and B also buy C". Usually, the classification hierarchy over the data items is available. Users are interested in generalized association rules that span different levels of the hierarchy, since sometimes more interesting rules can be derived by taking the hierarchy into account[Srikant and Agrawal, 1995, Han and Fu, 1995]. In our previous study, we proposed parallel algorithms and the candidate duplication based load balancing strategies for mining generalized association rules and evaluated their performance on 16-node shared-nothing parallel machine[Shintani and Kitsuregawa, 1998]. In [Shintani and Kitsuregawa, 1998], we showed that our algorithms could attain good performance on 16 nodes system. However, as the number of processor increase, it would be difficult to achieve flat workload distribution.

In this paper, we present the candidate partition based load balancing strategy for parallel algorithms of generalized association rule mining. This strategy partitions the candidate itemsets so that the number of candidate probes for each node is equalized each other with estimated support count by the information of previous pass. In our previous study, we proposed the candidate duplication based load balancing strategies, in which the workload is not considered at candidate partitioning. Moreover, we implement the parallel algorithms and the load balancing strategies for mining generalized association rules on a cluster of 100 PCs

interconnected with an ATM network, and analyze the performance of our algorithms using a large amount of transaction dataset. In [Han *et al.*, 1997], the parallel algorithms for mining flat association rules are experimented on 128 processor system. However, the transaction data is not read from actual disk in the experiments. In that experiments, the small transactions are kept in the buffer, and the transactions are read from the buffer instead of the actual disks. The size of transaction data does not exceed 50MBytes. On the other hand, the transactions are read from the actual disk and used a large amount of transactions (1GBytes) in our experiments.

This paper is organized as follows. In next section, we explain the parallel algorithms for mining generalized association rules. In section 3, we present load balancing strategies for parallel generalized association rule mining algorithm. Performance evaluations are given in section 4. Section 5 concludes the paper.

2 Parallel generalized association rule mining

First we introduce some basic concepts of generalized association rules presented in [Srikant and Agrawal, 1995]. Let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of items. Let $\mathcal{T}$ be a classification hierarchy on the items, which organize relationships of items in a tree form, shown in Figure 1. An edge in $\mathcal{T}$ represents an *is-a* relationship. Let $\mathcal{D} = \{t_1, t_2, \ldots, t_n\}(t_i \subseteq \mathcal{I})$ be a set of transactions, where each transaction t has an associated unique identifier called TID. We say a transaction t *contains* a set of items X, if X is a subset of t and the ancestor of items in t. The itemset X has *support* s in the transaction set $\mathcal{D}$, if s% of transactions in $\mathcal{D}$ contain X, here we denotes $s = supp(X)$. An *generalized association rules with classification hierarchy* is an implication of the form $X \Rightarrow Y$, where $X, Y \subset \mathcal{I}$, $X \cap Y = \phi$ and no item in Y is an ancestor of any item in X. Each rule has two measures of value, *support* and *confidence*. The *support* of the rule $X \Rightarrow Y$ is $supp(X \cup Y)$. The *confidence* c of the rule $X \Rightarrow Y$ in the transaction set $\mathcal{D}$ means c% of transactions in $\mathcal{D}$ that contain X also contain Y, which can be written as the ratio $supp(X \cup Y)/supp(X)$. Here a rule $x \Rightarrow ancestor(x)$ is redundant, since its confidence is always 100%.

The problem of mining generalized association rules is to find all the rules that satisfy a user-specified minimum support(min_supp) and minimum confidence (min_conf) on the assumption that we are given a set of transactions $\mathcal{D}$ and a classification hierarchy over the items. This problem can be decomposed into two subproblems:

1. Find all itemsets that have support above the user-specified minimum support. These itemsets are called the *large itemsets* and the other itemsets are called *small itemsets*.

2. For each large itemset, derive all rules that have more than user-specified minimum confidence as follows: for large itemset X and any Y $(Y \subset X)$, if $supp(X)/supp(X - Y) \geq min\_conf$, then the rule $(X - Y) \Rightarrow Y$ is derived.

The second subproblem, which derive the association rules, is processed in a straightforward manner. However, because of the large size of transaction data sets used in data mining, the first subproblem, which requires scanning the database, is a nontrivial problem. Most of association rule mining research focus this first subproblem.

Here we explain the Cumulate algorithm for finding all large itemsets, proposed in [Srikant and Agrawal, 1995]. First, the Cumulate algorithm generates candidate itemsets, then scans the transition database to determine whether the candidate itemsets satisfy the user specified minimum support. In the first pass (pass 1), support count for each item is counted by scanning the transaction database. All the items which satisfy the minimum support are picked out. These items are called large item (L_1). Hereafter k-itemset is defines a set of k items. The second pass (pass 2), the 2-itemsets are generated using L_1 which is called the candidate 2-itemsets (C_2), and delete any candidate in C_2 that consists of an item and its ancestor. Note that we need not count any itemset which contains both an item and its ancestor. Then the support count of C_2 is counted by scanning the transaction database. At the end of scanning the transaction data, the large 2-itemsets (L_2) which satisfy minimum support are determined. The following pass to find the large k-itemset is as described bellow.

1. Generate candidate itemsets:
   The candidate k-itemsets (C_k) are generated using large $(k-1)$-itemsets (L_{k-1}) as follows: join L_{k-1} with L_{k-1} and delete all the k-itemsets whose some of the $(k-1)$-itemsets are not in L_{k-1}. If k is 2, delete any candidates in C_2 that consists of an item and its ancestor.

2. Count support:
   Read the transaction database, add all ancestors of the items in a transaction t that are present in C_k.

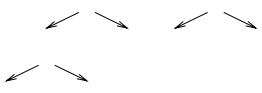


Figure 1: The classification hierarchy

Increment the support count of the candidates in C_k that are contained in t.

3. Determine large itemsets:
   The candidate itemsets in C_k are checked for whether they satisfy the minimum support or not, then the large k-itemsets (L_k) are determined.

This procedure terminates when the large itemset becomes empty.

2.1 Parallel algorithms

In this section, we describe parallel algorithms on shared-nothing parallel machines, NPGM(Non Partitioned Generalized association rule Mining) and H-HPGM(hash)(Hierarchical Hash Partitioned Generalized association rule Mining), proposed in [Shintani and Kitsuregawa, 1998].

2.2 Non Partitioned Generalized association rule Mining : NPGM

If the size of all the candidate itemsets is smaller than the size of the memory of each node, all the nodes can hold whole candidate itemsets. In such a case, parallelization is straightforward. By partitioning the transaction database over all the nodes, the transaction data can be read and candidate itemsets can be counted in parallel. In NPGM, the candidate itemsets are copied over all the nodes, each node can work independently and the final statistics are gathered into a coordinator node where minimum support conditions are examined. The procedure of pass k is as follows.

1. Generate candidate itemsets:
   Each node generates C_k using L_{k-1}. If k is 2, delete the candidates that contains an items and its ancestor.

2. Count support:
   Each node reads the transaction database from its local disk, generates extended transaction t' by adding all ancestors of the items in a transaction t that are present in C_k.
   Increment the support count of the candidates in C_k that are contained in t'.

3. Determine large itemsets:
   After reading all the transaction data, all node's support count are gathered into the coordinator node and checked to determine whether the minimum support condition is satisfied or not.

If the size of all the candidate itemsets exceeds the local memory of a single node, the candidate itemsets are partitioned into fragments, each of which can fits within the local memory of a single node, and the above process is repeated for each fragment. The disk I/O becomes prohibitively costly when the candidate itemsets becomes large.

2.3 Hierarchical Hash Partitioned Generalized association rule Mining : H-HPGM(hash)

H-HPGM(hash) partitions the candidate itemsets among the nodes taking the classification hierarchy into account so that all the candidate itemsets whose root items are identical be allocated to the same node, which eliminates communication of the ancestor items. Thus the communication overhead can be kept low. The procedure of pass k is as follows.

1. Generate candidate itemsets:
   Each node generates C_k in the same way as NPGM. For each candidate, the destination node ID is determined by applying the hash function to replacing each item of the candidate itemset with their root items. If the ID is its own, insert it into the candidate table(C_k^n).

2. Count support:
   Each node reads the transaction database from its local disk and generates extended transaction t' by replacing the item in t with the large item in its ancestors which is closest to the bottom, if there are small items. For each node n, select the related items from t' and send them to n-th node. For the itemsets received from other nodes and those locally generated, generate k-itemset from the itemsets and increment the support count of this k-itemset and its all ancestor candidates.

3. Determine large itemsets:
   After reading all the transactions, each node can determine the large itemset in C_k^n. The coordinator node gather all the large k-itemset.

3 Load balancing strategy

In this section, we present two kinds of load balancing strategies, the candidate partition based strategy and the candidate duplication based strategy. The candidate partition based strategy equalizes the number of candidate probes with estimated support count by the statistics of data. The candidate duplication based strategy, which was proposed in [Shintani and Kitsuregawa, 1998], duplicates the frequently occurring candidate itemset among all the nodes.

3.1 Candidate partition based load balancing strategy

In H-HPGM(hash), each node probes the itemsets generated with received itemset against its own candidate table. The number of candidate probes is associated with the support of assigned candidate itemsets. This means that the workload depends on the assigned candidate itemsets. Some itemsets have higher support value, which cause a large number of candidate probes. In our

skew handling strategy, we assign the candidate itemsets so that the number of candidate probes for each nodes be equal each other. Here, we have to set the weighting factor for each itemset. Since real support value is attained after the execution, the exact weighting factor is not available before the execution. In our skew handling strategy, we set the weighting factor from the statistics of data obtained at previous pass. H-HPGM(hash) algorithm is consisted with several passes. For each pass, the transaction database is scanned and the large k-itemsets are determined. At pass k, we can utilize the support value of large $(k-1)$-itemset. Suppose X is a candidate itemset at pass k, the support count of all the size-$(k-1)$ subsets of X are available. The upper bound of the support value of X is the minimum value of the support count of all the size-$(k-1)$ subsets of X, i.e., the maximum value of support value of itemset X is defined as follows[Cheung *et al.*, 1996a, Cheung *et al.*, 1996b].

$$\begin{aligned} & max\_supp(X) \\ & = min\{supp(Y) \mid Y \subset X, \ and \ |Y| = k-1\} \end{aligned} \quad (1)$$

Here, $supp(X)$ means the support of itemset X. Thus, we estimate the support of candidate k-itemset using the support of large $(k-1)$-itemsets and employ them to set the weighting factor.

3.1.1 H-HPGM with Statistics: H-HPGM(stat)

H-HPGM(hash) partitions the candidate itemsets among the nodes so that all the candidate itemsets whose root items are the same be allocated to the same node. That is, H-HPGM(hash) divides the candidate itemsets into the hierarchy of the candidate itemsets and allocates such whole hierarchy to a node. Thus the granule is a hierarchies, that is, a tree.

We set the weighting factor for the granule of candidate partition, that is, the combination of trees. The weighting factor of the combination of trees X is defined by following.

$$W(X) = \sum_{y \in Y} max\_supp(y) \quad (2)$$

Here, $W(X)$ means the weighting factor of X, Y means all the descendant candidate itemsets of X. For example, we consider that the taxonomy of items at pass 2 is given in Figure 2. The weighting factor of the combination of trees $\{1,1\}$ and $\{1,2\}$ are calculated as follows.

$$\begin{aligned} & W(\{1,1\}) \\ & = min\{supp(\{4\}), supp(\{5\})\} \\ & \quad + min\{supp(\{5\}), supp(\{10\})\} \end{aligned}$$

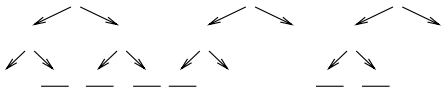


Figure 2: Taxonomy at pass 2

(1) **foreach** size-k combination of root items x **do**
(2) Select the minimum node n whose weight is smallest of all the nodes
(3) $C_k^n := x$ and all descendant candidates of x (C_k^n means the set of candidates allocated to n-th node)
(4) $CW(n) +=$ weight of x ($CW(n)$ means the sum of weight of allocated candidates to n-th node)
(5) **end**

Figure 3: The procedure to allocate the size-k combination of root items

$$\begin{aligned} & W(\{1,2\}) \\ & = min\{supp(\{1\}), supp(\{2\})\} \\ & \quad + min\{supp(\{1\}), supp(\{6\})\} \\ & \quad + min\{supp(\{1\}), supp(\{7\})\} \\ & \quad \cdots \\ & \quad + min\{supp(\{7\}), supp(\{10\})\} \\ & \quad + min\{supp(\{10\}), supp(\{15\})\} \end{aligned}$$

The procedure to calculate the weighting factor and to allocate the candidate itemsets among the nodes at pass k are as follows.

1. Generate the size-k combination of root items.
2. Calculate the weighting factor of the size-k combination of root items using the equation (2).
3. Sort the size-k combination of root items based on their weight.
4. The size-k combination of root items are allocated among the nodes so that the sum of weighting factor of allocated size-k combination is equalized for each node (Figure 3).

3.2 Candidate duplication based load balancing strategy

In the case that the size of the candidate itemsets is smaller than the available system memory, H-HPGM(hash) and H-HPGM(stat) do not use the remaining free space. If the transaction data is skewed, that is, there are some itemsets which appear very frequently in the transaction data, the node which is allocated such itemsets will receive a lot of transaction

data, which incurs a system bottleneck. The candidate duplication based strategies handle this problem by identifying such frequently occurring itemsets, duplicating them among all the nodes and counting the support locally. The duplicated candidates are processed in the same way as NPGM. The remaining candidates are partitioned and processed in the same way as H-HPGM(stat).

In the candidate duplication based load balancing strategy, the effect increase as the size of duplicated candidate itemsets increase. In order to attain both flat workload distribution and flat distribution of the number of candidate itemsets among the nodes, we modify the equation of setting the weighting factor as follows.

$$W(X) = \frac{\sum_{y \in Y} max\_supp(y)}{\#\ of\ transactions} + \frac{CN(X)}{CN_k - CN^D} \quad (3)$$

Here, $CN(X)$ means the number of descendant candidate itemsets of X, CN_k means the number of candidate itemsets at pass k, CN^D means the number of duplicated candidate itemsets. Y means the descendant candidate itemsets of X excluding the duplicated candidate itemsets.

We named the load balancing strategy using the equation(3) H-HPGM(stat+). The procedure to allocate the partitioning candidate itemsets among the nodes is obtained by replacing the equation (2) in step 2 with equation (3) in the procedure of H-HPGM(stat).

1. Same as H-HPGM(stat).
2. Calculate the weighting factor of the size-k combination of root items using the equation (3)
3. – 4. Same as H-HPGM(stat).

For example, we consider that the taxonomy of items at pass 2 is given in Figure 2. Assume the number of duplicated candidate itemsets is 0. The weighting factor of the combination of trees $\{1,1\}$ and $\{1,2\}$ are calculated as follows.

$$\begin{aligned} & W(\{1,1\}) \\ & = \{min\{supp(\{4\}), supp(\{5\})\} \\ & \quad + min\{supp(\{5\}), supp(\{10\})\}\} \\ & \quad / \{\#\ of\ transactions\} \\ & \quad + \{2\ /\ 45\} \end{aligned}$$

$$\begin{aligned} & W(\{1,2\}) \\ & = \{min\{supp(\{1\}), supp(\{2\})\} \\ & \quad + min\{supp(\{1\}), supp(\{6\})\} \\ & \quad + min\{supp(\{1\}), supp(\{7\})\} \\ & \quad \ldots \\ & \quad + min\{supp(\{7\}), supp(\{10\})\} \\ & \quad + min\{supp(\{10\}), supp(\{15\})\}\} \\ & \quad / \{\#\ of\ transactons\} \\ & \quad + \{16\ /\ 45\} \end{aligned}$$

3.2.1 H-HPGM(stat+) with Tree Grain Duplicate: H-HPGM-TGD(stat+)

H-HPGM-TGD(stat+) detects the tree whose candidate itemsets contain frequently occurred items, duplicates them among the nodes and counts the support count locally for those itemsets like in NPGM. The procedure to detect the duplicated candidates in pass k is as follows.

1. Count up the number of candidates allocated for each node by generating the k-itemsets using L_{k-1} and calculate the size of free memory space.
2. Count the number of descendant candidates for each root k-items combination.
3. Generate k-items combination from root items. Here, these k-items combination contain the items consisting of the same items, such as $\{1,1\}$.
4. Sort the combination of root items based on those item's frequency of appearance.
5. Choose the most frequently occurring combination of root itemsets and duplicate them and their descendant candidate itemsets among the nodes.
6. The remaining candidate itemsets are partitioned in the same way as H-HPGM(stat+).

3.2.2 H-HPGM(stat+) with Path Grain Duplicate: H-HPGM-PGD(stat+)

H-HPGM-PGD(stat+) picks up the leaf large items and sorts them based on their support value. Then it chooses the most frequently occurring itemsets and copies them and their all ancestor itemsets over all the nodes. Since the granule of candidate duplication employed in H-HPGM-PGD(stat+) is smaller than H-HPGM-TGD(stat+), it can balance the load among the nodes more effectively. The procedure to detect the duplicated candidates in pass k is as follows.

1. – 3. Same as H-HPGM-TGD(stat+).
4. Pick up the large items in L_{k-1} which is the closest to the bottom, and sort them based on their support value.
5. Choose the first several most frequently occurring items using the sorted list derived at "4", and duplicate it and its all ancestor candidates among the nodes.
6. Same as H-HPGM-TGD(stat+).

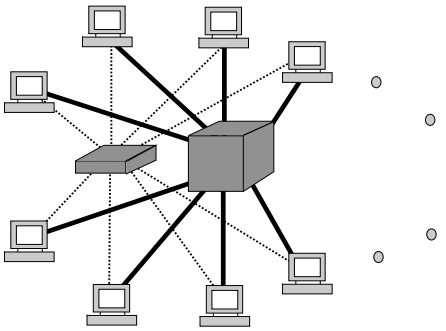


Figure 4: An overview of the PC cluster

3.2.3 H-HPGM(stat+) with Fine Grain Duplicate: H-HPGM-FGD(stat+)

H-HPGM-FGD(stat+) checks the frequently occurring itemsets which consists of the any level items. It duplicates them and their all ancestor itemsets over all the nodes. Thus only the frequent candidate itemsets are duplicated. The granule of candidate duplication becomes finer. The procedure to detect the duplicated candidates in pass k is as follows.

1. – 3. Same as H-HPGM-TGD(stat+).
4. Sort the large items based on their count support value.
5. Choose the first most frequently occurring candidate itemsets, and duplicate them and their all ancestor candidates among the nodes.
6. Same as H-HPGM-TGD(stat+).

4 Large scale PC cluster system

4.1 Components of cluster

Our PC cluster system[Tamura *et al.*, 1997] consists of one hundred 200MHz Pentium Pro PCs, connected with an 155Mbps ATM switch as well as by 10Mbps Ethernet network. Figure 4 shows an overview of the PC cluster. Each node consists of components shown in Table 1. We use the RFC-1483 PVC driver for IP over ATM. TCP/IP is used as a communication protocol. HITACHI's AN1000-20, which has 128 ports, is used as an ATM switch.

5 Performance analysis

We implement the parallel algorithms and the load balancing strategies on the PC cluster system. The transaction database is evenly partitioned over the local disk of all the nodes. Solaris socket library is used for the inter-process communication. All processes are connected with each other by socket connections, thus forming mesh topology. To evaluate the performance of the proposed parallel algorithms, synthetic dataset emulating retail transactions is used. The generation procedure is described in [Srikant and Agrawal, 1995]. Table 2 shows the meaning of the various parameters and the characteristics of the dataset used in our experiment.

5.1 Execution time

We show the number of candidate itemsets and large itemsets, and the execution time at each pass in Table 3. 64 nodes in PC cluster system are used, and the minimum support is set to 0.3% in this experiments.

NPGM can attain the best performance when the number of candidate itemsets is small, since NPGM does not require the communication of transaction data for the support count process. When the number of candidate itemsets becomes large such as pass 2, the single node's memory cannot hold the entire candidate itemsets, which degrade the performance. H-HPGM(hash) and H-HPGM(stat) partition the candidate itemsets among the nodes. These candidate partition based algorithms outperform than NPGM when the number of candidate is large. By the candidate partition based load balancing strategy, H-HPGM(stat) achieves better performance than H-HPGM(hash) at all the range of minimum support. When the number of candidate itemsets is small, the performance of H-HPGM(hash) and H-HPGM(stat) is worse than NPGM. H-HPGM(hash) and H-HPGM(stat) partition the candidate itemsets at all the pass. When the number of candidate becomes small compared with the number of nodes, the allocated candidate itemsets for each node decrease, the most of memory space does not utilized. H-HPGM-FGD(stat+) can archive good performance at all the passes. In the case that each node's memory can hold the entire candidate itemsets, H-HPGM-FGD(stat+) behaves the same way as NPGM. Because of some overhead, the execution time of H-HPGM-FGD(stat+) is a little longer than that of NPGM when the number of candidate itemsets is small. The whole execution time of H-HPGM-FGD(stat+) is best.

In Figure 5, we show the execution time at pass 2 varying the minimum support. Hereafter, we show only the result at pass 2, since the number of candidate itemsets and the execution time for pass 2 is longest. 64 nodes in PC cluster system are used in this experiment. We show the result of the best workload distribution, named Flat, as an ideal result. By using the real support count information to set the weighting factor, we can attain the most flat workload distribution.

The execution time of all the algorithms increases when the minimum support becomes small. Especially,

Table 1: Each node of PC cluster

CPU	Intel 200MHz Pentium Pro
Chipset	Intel 440FX
Main memory	64MBytes
Disk drive	For OS Western Digital Caviar 32500 (EIDE, 2.5GB)
	For database Seagate Barracuda (Ultra SCSI, 4.3GB)
OS	Solaris2.5.1 for x86
ATM NIC	Interphase 5515 PCI ATM Adapter

Table 2: Parameters of dataset

Parameter	
Number of transactions	20,000,000 (file size 1GB)
Average size of the transactions	5
Average size of the maximal potentially large itemsets	5
Number of maximal potentially large itemsets	10,000
Number of items	50,000
Number of roots	100
Number of levels	5–6
Fanout	5

Table 3: # of candidate itemsets and large itemsets, and execution time at each pass

Pass	C	L	Execution time (sec)			
			NPGM	H-HPGM (hash)	H-HPGM (stat)	H-HPGM-FGD(stat+)
1	50000	1361	3.4	3.4	3.4	3.4
2	881548	4188	592.8	272.9	178.5	153.9
3	31404	970	116.2	195.1	157.4	118.1
4	3153	176	69.3	132.6	101.1	70.9
5	927	86	52.8	89.7	82.0	53.4
6	322	5	13.6	31.1	28.5	13.7
Total			848.1	724.8	550.9	413.4

C : Number of candidate itemsets
L : Number of large itemsets

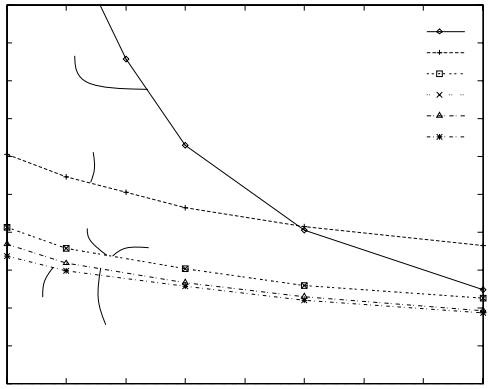


Figure 5: Execution time

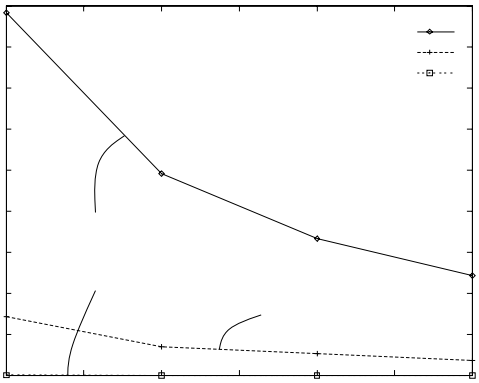


Figure 6: The distribution of candidate itemsets at pass 2

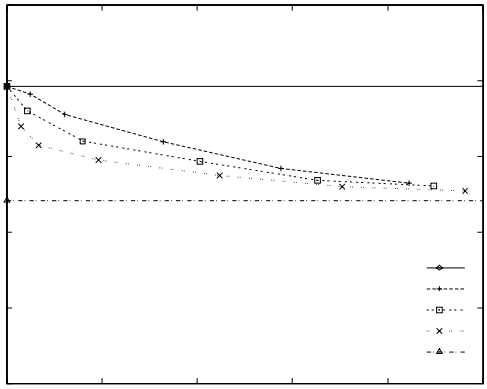


Figure 7: Execution time with varying the size of duplicated candidates

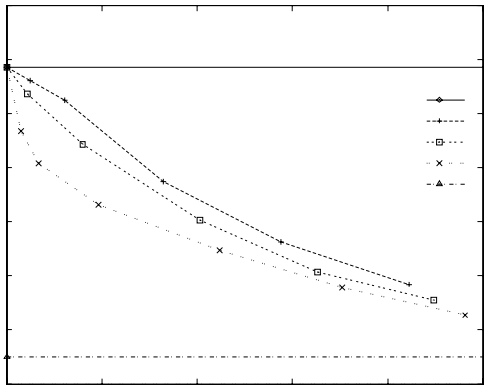


Figure 8: The number of candidate probes at pass 2

the execution time of NPGM increases sharply. When the minimum support reduces, the number of candidate itemsets increases. When the minimum support is small, the candidate partitioned methods can attain good performance. The performance of H-HPGM(stat) and H-HPGM(stat+) is almost equal. H-HPGM-FGD(stat+) attains the best performance of all the range of minimum support.

5.2 The distribution of allocated candidate itemsets for each node

Figure 6 show the distribution of allocated candidate itemsets for each node at pass 2. The vertical axis is the standard deviation of the number of candidate itemsets for each node. The minimum support is varying from 0.2% to 0.5%.

H-HPGM(stat+) attains flat candidate distribution. Since H-HPGM(stat+) sets the weighting factor considering not only the distribution of workload but also the distribution of the number of candidate itemsets among the nodes, it can attain flat distribution of candidate itemsets.

5.3 Performance with varying the size of duplicated candidates

Here, we show the performance of load balancing strategies with varying the size of duplicated candidate itemsets at pass 2. Figure 7 shows the execution time. Figure 8 shows the workload distribution. Figure 9 shows the percentage of candidate probes for duplicated candidate itemsets to all the candidate probes. Figure 10 shows the average amount of received messages for each node. In all the experiments, the minimum support is set to 0.3% and 64 nodes of PC cluster system is activated.

As the size of duplicated candidate itemsets increases, the execution time of candidate duplication based load balancing strategy is reduced and the workload distribution becomes balanced. The candidate duplication based load balancing strategies detect the frequently oc-

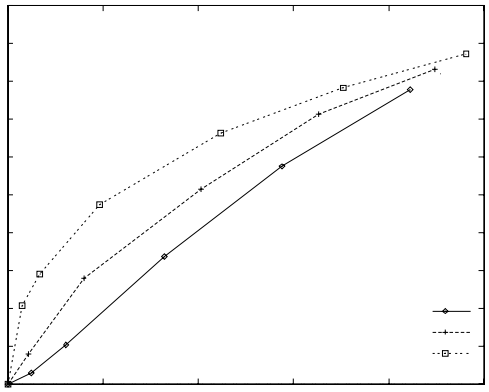


Figure 9: Percentage of candidate probes for duplicated candidates at pass 2

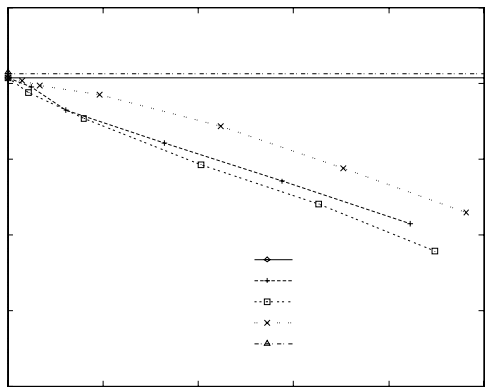


Figure 10: Average amount of received messages at pass 2

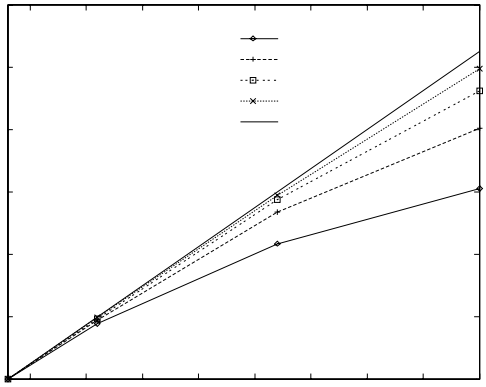


Figure 11: Speedup ratio at pass 2

curring candidate itemsets and duplicate them so that the remaining free space can be utilized as much as possible. Especially, H-HPGM-FGD(stat+) can reduce the execution time at small candidate duplication size. Since H-HPGM-FGD(stat+) employs the smallest grain duplication strategy, it can show the ability if the remaining free memory space is small. Figure 9 shows that the small size of candidate duplicates occupies the large number of candidate probes in the candidate duplication based load balancing strategies. In the candidate duplication based load balancing strategies, we duplicate the frequently occurring candidate itemsets. Support count process for duplicated candidate itemsets can be locally processed like as NPGM, which reduce the communication overhead.

5.4 Speedup

Figure 11 shows the speedup ratio with varying the number of nodes used 16, 32, 64 and 100. The curves are normalized by the execution time of 16 nodes system. Here, the minimum support is set to 0.3%.

The load balancing strategies attain higher linearity than H-HPGM(hash). Since H-HPGM(hash) partitions the candidate itemsets using hash function without considering the load balancing and duplicates no candidate itemsets, the workload skew degrades the linearity. Though H-HPGM(stat) partitions the candidate itemsets taking the load balancing into account, it cannot attain sufficient linearity. On the other hand, H-HPGM-FGD(stat+), duplicating the frequently occurring candidate itemsets and partitioning the other candidate itemsets with considering the load balance, can attain the highest linearity. In Figure 11, H-HPGM-FGD(stat) achieves good performance on one hundred nodes system. By considering both the distribution of workload and the number of candidate itemsets, we can attain good performance.

6 Conclusions

In this paper, we presented the candidate partition based load balancing strategy for parallel mining algorithm of generalized association rule mining and evaluated their performance through the implementation on large scale PC cluster system.

H-HPGM(hash) partitions the candidate itemsets using hash function without considering the load balance and duplicates no candidate itemsets, it cannot attain flat workload distribution. H-HPGM(stat) partitions the candidate itemsets so that the number of candidate probes for each node is equalized each other with estimated support count by the information of previous pass. This estimated support count might contain some error, but it attain better performance than H-HPGM(hash) which using no weighting factor. However, as the number of processor increase, it would be difficult to achieve sufficient flat workload distribution. H-HPGM-TGD(stat+), H-HPGM-PGD(stat+) and H-HPGM-FGD(stat+) combine the candidate partition based load balancing strategy and the candidate duplication based load balancing strategy. Support counting for duplicated candidate itemsets can be locally processed which reduce the communication overhead and the workload skew.

In this paper, we examined the effectiveness of parallel algorithms and their load balancing strategies on large scale parallel computer system using the large amount of transaction dataset. Our system is consisted with one hundred PC's and 1GBytes transaction database was used for experiment. As far as the authors know, there has no research on parallel data mining over such large scale systems using such large amount of transaction database. Through several experiments, we showed H-HPGM-FGD(stat+) could attain sufficiently good performance and achieve good workload distribution on one hundred PC cluster system.

References

[Cheung *et al.*, 1996a] D.W. Cheung, J. Han, V.T. Ng, A.W. Fu, and Y. Fu. A fast distributed algorithm for mining association rules. In *Proceedings of 4th International Conference on Parallel and Distributed Information Systems*, pages 31–42, December 1996.

[Cheung *et al.*, 1996b] D.W. Cheung, V.T. Ng, A.W. Fu, and Y. Fu. Efficient mining of association rules in distributed databases. In *IEEE Transactions on Knowledge and Data Engineering, Vol.8, No.6*, pages 911–922, December 1996.

[Han and Fu, 1995] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proceedings of 21th International Conference on Very Large Data Bases*, pages 420–431, 1995.

[Han *et al.*, 1997] E.-H. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. In *Proceedings of 1997 ACM SIGMOD International Conference on Management of Data*, pages 277–288, 1997.

[Shintani and Kitsuregawa, 1998] T. Shintani and M. Kitsuregawa. Parallel algorithms for mining generalized association rules with classification hierarchy. In *Proceedings of 1998 ACM SIGMOD International Conference on Management of Data*, pages 25–36, June 1998.

[Srikant and Agrawal, 1995] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proceedings of 21th International Conference on Very Large Data Bases*, pages 407–419, 1995.

[Tamura *et al.*, 1997] T. Tamura, M. Oguchi, and M. Kitsuregawa. Parallel database processing on a 100 node pc cluster: Cases for decision support query processing and data mining. In *Proceedings of Supercomputing 97::High Performance Networking and Computing*, 1997.

Integrated Delivery of Large-Scale Data Mining Systems[1]

Extended Abstract

Graham J. Williams
Cooperative Research Centre for Advanced Computational Systems
CSIRO Mathematical and Information Sciences
GPO Box 664 Canberra 2601 Australia
Email: Graham.Williams@cmis.csiro.au

Introduction

Data Mining draws on many technologies to deliver novel and actionable discoveries from very large collections of data. The Australian Government's Cooperative Research Centre for Advanced Computational Systems (ACSys) is a link between industry and research focusing on the deployment of high performance computers for data mining (as well as virtual environments, interactive media, and advanced servers). The multidisciplinary ACSys team draws together researchers in Statistics, Machine Learning, and Numerical Algorithms from The Australian National University and the Australian Government's research organisation CSIRO. Commercial projects from banking, insurance, and health guide our research.

We present an overview of the work of the ACSys Data Mining projects where the use of large-scale, high performance, computers plays a key role. We highlight the use of large-scale computing within three complimentary areas: parallel algorithms for data analysis, virtual environments for data mining, and data management for data mining. The Data Miner's Arcade provides simple abstractions to integrate these components providing high performance data access for a variety of data mining tools communicating through XML.

Parallel Algorithms

Algorithms used in data mining projects include generalised additive models, thin plate splines, decision tree and rule induction, patient rule induction methods, and more recently combination of simple rules. The issue of scalability must be addressed in the context of data mining. We illustrate this with an extension to multivariate adaptive regression splines using BMARS. MARS (Friedman 1991) constructs a linear combination of basis functions which are products of one-dimensional basis functions (indicator functions in the case of categorical variables and truncated power functions in the case of numeric variables). The key to the method is that the basis functions are generated recursively and depend on the data. The important implication of the approach is that models produced by MARS involve only variables and their interactions relevant to the problem at hand. This property of MARS models is especially useful in a data mining context. BMARS (Bakin, Hegland, Howard and Williams 1999) improves upon MARS by: using compactly supported B-spline basis functions; utilising a new scale-by-scale model building strategy; and introducing a parallel implementation. These modifications allow the stable and *fast* (compared to MARS) analysis of very large datasets.

Virtual Environments

All stages of a data mining project require considerable understanding of multidimensional data. Visualisation tools, both for exploratory data analysis and for exploring the models produced by the data analysis algorithms, can play a significant role, particularly in the context of complex models generated through data mining (Williams and Huang 1997, Williams 1999). Traditional approaches tend to be limited by the mouse-keyboard-monitor interface. Virtual environments (VEs) dramatically increase the "canvas" on which to render graphic representations of the data that scale to large numbers of dimensions through an interactive, immersive, environment.

An approach being explored for this task is a technique for partitioning a 3D VE into smaller working regions, each of which is capable of holding a subspace of the original multidimensional data space (Nagappan 1999). The algorithm distributes a set of partitioning axes in a radial arrangement from a single common origin, with one axis for each dimension in the data set. The ends of the axes thus lie on the surface of a sphere. A convex hull is generated to connect the ends of the axes together. The axes and the space that they form can be used for a number of visualisation strategies, including rectangular prism and the use of density functions.

[1]The author acknowledges the support provided by the Cooperative Research Centre for Advanced Computational Systems (ACSys) established under the Australian Government's Cooperative Research Centres Program.

Data Management

Data is stored in a variety of formats and within a variety of database systems, and needs to be accessed in a timely manner and potentially multiple times. Smart caching and other optimisations, tuned for particular classes of analysis algorithms, is required. The semantic extension framework (SEF) for Java (Marquez, Zigman and Blackburn 1999) is an abstraction tool which provides orthogonality of the algorithms with respect to the data source. This approach allows datasets to be transparently accessed and efficiently managed from any source. Algorithms accessing the data simply view the data as Java data structures which are efficiently instantiated as required and as determined by the semantic extensions provide for the relevant objects. This new project is exploring the use of the SEF to provide orthogonality and optimised access to large scale datasets.

Pulling it Together

The Java-based Data Miner's Arcade (Williams 1998) is a platform-independent system for integrating multiple analysis and visualisation tools with a consistent user interface, providing seamless access to data stored in a multitude of systems. Standard interfaces are used where available and data is accessed through ODBC and JDBC or from other sources and managed internally within the Arcade. This is proposed to be redeveloped using the Java semantic extension framework to provide orthogonality between the analysis algorithms and the data sources. The Extensible Markup Language (XML) is adopted as the target "language" for the data mining tools within the environment (Grossman, Bailey, Ramu, Malhi, Hallstrom, Pulleyn and Qin 1999). Models expressed in XML can be visualised, run, or combined with other models in ensemble systems, through the use of plug-ins within the Data Miner's Arcade.

Acknowledgments

The worked reported on here has been performed by the ACSys project. Contributions from both the Australian National University and CSIRO Australia staff are acknowledge. In particular, the BMARS work was Dr Sergey Bakin's PhD topic, the specific visualisation approach mentioned is the work of Raj Nagappan, and the data management work relies on the research of Dr Stephen Blackburn and Dr Alonzo Marquez. Dr Markus Hegland, Peter Milne, and Dr Stephen Roberts have also made many contributions.

References

Bakin, S., Hegland, M., Howard, P. and Williams, G.: 1999, Mining taxation data with parallel BMARS, *Submitted for publication* .

Friedman, J.: 1991, Multivariate adaptive regression splines, *The Annals of Statistics* **19**(1), 1–141.

Grossman, R., Bailey, S., Ramu, A., Malhi, B., Hallstrom, P., Pulleyn, I. and Qin, X.: 1999, The management and mining of multiple predictive models using the predictive modelling markup language, *Information and Software Technology* **41**(9).

Marquez, A., Zigman, J. and Blackburn, S.: 1999, Fast, portable orthogonally persistent java using semi dynamic semantic extensions, *Submitted for publication* .

Nagappan, R.: 1999, Visualising multidimensional non-geometric data sets, *Submitted for publication* .

Williams, G.: 1998, The data miner's arcade, http://www.cmis.csiro.au/Graham.Williams/dataminer/Arcade.html.

Williams, G. J.: 1999, Evolutionary hot spots data mining, *Advances in Data Mining (PAKDD99)*, Lecture Notes in Computer Science, Springer-Verlag.

Williams, G. J. and Huang, Z.: 1997, Mining the knowledge mine: The Hot Spots methodology for mining large, real world databases, *in* A. Sattar (ed.), *Advanced Topics in Artificial Intelligence (AI97)*, Vol. 1342 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 340–348.

A Data-Clustering Algorithm On Distributed Memory Multiprocessors*

Inderjit S. Dhillon and Dharmendra S. Modha
IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120-6099
emails: {dhillon,dmodha}@almaden.ibm.com

Abstract

To cluster increasingly massive data sets that are common today in data and text mining, we propose a parallel implementation of the k-means clustering algorithm based on the message passing model. The proposed algorithm exploits the inherent data-parallelism in the k-means algorithm. We analytically show that the speedup and the scaleup of our algorithm approach the optimal as the number of data points increases. We implemented our algorithm on an IBM POWERparallel SP2 with a maximum of 16 nodes. On typical test data sets, we observe nearly linear relative speedups, for example, 15.62 on 16 nodes, and essentially linear scaleup in the size of the data set and in the number of clusters desired. For a 2 gigabyte test data set, our implementation drives the 16 node SP2 at more than 1.8 gigaflops.

Keywords: k-means, data mining, massive data sets, message-passing, text mining.

1 Introduction

Data sets measuring in gigabytes and even terabytes are now quite common in data and text mining, where a few million data points are the norm. For example, the patent database (www.ibm.com/patents/), the LexisNexis document collection containing more than 1.5 billion documents (www.lexisnexis.com), and the Internet archive (www.alexa.com) are in multi-terabyte range. When a sequential data mining algorithm cannot be further optimized or when even the fastest available serial machine cannot deliver results in a reasonable time, it is natural to look to parallel computing. Furthermore, given the monstrous sizes of the data sets, it often happens that they cannot be processed *in-core*, that is, in the main memory of a single processor machine. In such a situation, instead of implementing a disk based algorithm which is likely to be considerably slower, it is appealing to employ parallel computing and to exploit the main memory of *all* the processors.

Parallel data mining algorithms have been recently considered for tasks such as association rules and classification, see, for example, [Agrawal and Shafer1996], [Chattratichat *et al.*1997], [Cheung and Xiao1999], [Han *et al.*1997], [Joshi *et al.*1998], [Kargupta *et al.*1997], [Shafer *et al.*1996], [Srivastava *et al.*1998], [Zaki *et al.*1998], and [Zaki *et al.*1997]. Also, see [Stolorz and Musick1997] and [Freitas and Lavington1998] for recent books on scalable and parallel data mining.

In this paper, we consider parallel clustering. Clustering or *grouping of similar objects* is one of the most widely used procedures in data mining [Fayyad *et al.*1996]. Practical applications of clustering include unsupervised classification and taxonomy generation [Hartigan1975], nearest neighbor searching [Fukunaga and Narendra1975], scientific discovery [Smyth *et al.*1997], vector quantization [Gersho and Gray1992], time series analysis [Shaw and King1992], multidimensional visualization [Dhillon *et al.*1998], and text analysis and navigation [Cutting *et al.*1992].

As our main contribution, we propose a parallel clustering algorithm on distributed memory multiprocessors, that is, on a shared-nothing parallel machine, and analytically and empirically validate our parallelization strategy. Specifically, we propose a parallel version of the popular k-means clustering algorithm [Hartigan1975] based on the message-passing model of parallel computing [Gropp *et al.*1996, Snir *et al.*1997].

We now briefly outline the paper, and summarize our results. In Section 2, we present the k-means algorithm. In Section 3, we carefully analyze the computational complexity of the k-means algorithm. Based on this analysis, we observe that the k-means algorithm is inherently data-parallel. By exploiting this parallelism, we design a parallel k-means algorithm. We analytically show that the speedup and the scaleup of our algorithm approach the optimal as the number of data points increases. In other words, we show that as the number of data points increases the communication costs incurred by our parallelization strategy are relatively insignificant compared to the overall computational complexity. Our parallel algorithm is based on the message-passing model of parallel computing; this model is also briefly

*A version of this paper has been submitted to *IEEE Transactions on Knowledge and Data Engineering.*

reviewed in Section 3. In Section 4, we empirically study the performance of our parallel k-means algorithm (that is, speedup and scaleup) on an IBM POWERparallel SP2 with a maximum of 16 nodes. We empirically establish that our parallel k-means algorithm has nearly linear speedup, for example, 15.62 on 16 nodes, and has nearly linear scaleup behavior. To capture the effectiveness of our algorithm in a nutshell, note that we are able to to drive the 16 node SP2 at nearly 1.8 gigaflops (floating point operations) on a 2 gigabyte test data set.

2 The k-means Algorithm

Suppose that we are given a set of n data points $X_1, X_2, \cdots, X_n$ such that each data point is in R^d. The problem of finding the *minimum variance* clustering of this data set into k clusters is that of finding k points $\{m_j\}_{j=1}^{k}$ in R^d such that

$$\frac{1}{n}\sum_{i=1}^{n}\left(\min_{j} d^2(X_i, m_j)\right), \qquad (1)$$

is minimized, where $d(X_i, m_j)$ denotes the Euclidean distance between X_i and m_j. The points $\{m_j\}_{j=1}^{k}$ are known as *cluster centroids* or as *cluster means*. Informally, the problem in (1) is that of finding k cluster centroids such that the average squared Euclidean distance (also known as the mean squared error or MSE, for short) between a data point and its nearest cluster centroid is minimized. Unfortunately, this problem is known to be NP-complete [Garey *et al.*1982].

The classical k-means algorithm [Hartigan1975] provides an easy-to-implement approximate solution to (1). Reasons for popularity of k-means are ease of interpretation, simplicity of implementation, scalability, speed of convergence, adaptability to sparse data, and ease of out-of-core implementation [Zhang *et al.*1996]. We present this algorithm in Figure 1, and intuitively explain it below:

1. (**Initialization**) Select a set of k starting points $\{m_j\}_{j=1}^{k}$ in R^d (line 5 in Figure 1). The selection may be done in a random manner or according to some heuristic.

2. (**Distance Calculation**) For each data point X_i, $1 \leq i \leq n$, compute its Euclidean distance to each cluster centroid m_j, $1 \leq j \leq k$, and then find the closest cluster centroid (lines 14-21 in Figure 1).

3. (**Centroid Recalculation**) For each $1 \leq j \leq k$, recompute cluster centroid m_j as the average of data points assigned to it (lines 22-26 in Figure 1).

4. (**Convergence Condition**) Repeat steps 2 and 3, until convergence (line 28 in Figure 1).

The above algorithm can be thought of as a gradient-descent procedure which begins at the starting cluster centroids and iteratively updates these centroids to decrease the objective function in (1). Furthermore, it is known that k-means will always converge to a local minimum [Bottou and Bengio1995]. The particular local minimum found depends on the starting cluster centroids. As mentioned above, the problem of finding the global minimum is NP-complete.

Before the above algorithm converges, steps 2 and 3 are executed a number of times, say $\mathfrak{I}$. The positive integer $\mathfrak{I}$ is known as the *number of k-means iterations*. The precise value of $\mathfrak{I}$ can vary depending on the initial starting cluster centroids even on the same data set.

A completely different tree-structured algorithm for k-means has been presented by Alsabti, Ranka, and Singh [Alsabti *et al.*1998]. It is not clear, however, whether their algorithm is amenable to massive parallelization akin to ours.

Next, we analyze, in detail, the computational complexity of the above algorithm, and propose a parallel implementation.

3 Parallel k-means: Design and Analysis

Our parallel algorithm design is based on the Single Program Multiple Data (SPMD) model using message-passing which is currently the most prevalent model for computing on distributed memory multiprocessors. We assume a shared-nothing machine with P processors– each with a local memory. The communication between these processors is captured by MPI, the Message Passing Interface, which is a standardized, portable, and widely available message-passing system designed by a group of researchers from academia and industry [Gropp *et al.*1996, Snir *et al.*1997].

From a programmer's perspective, parallel computing using MPI appears as follows. The programmer writes a single program in C (or C++ or FORTRAN 77), compiles it, and links it using the MPI library. The resulting object code is loaded in the local memory of every processor taking part in the computation; thus creating P "parallel" *processes*. Each process is assigned a unique identifier between 0 and $P-1$. Depending on its processor identifier, each process may follow a distinct execution path through the same code. These processes may communicate with each other by calling appropriate routines in the MPI library which encapsulates the details of communications between various processors. Table 1 gives a glossary of various MPI routines which we use in our parallel version of k-means in Figure 2.

MPI_Comm_size()	returns the number of processes
MPI_Comm_rank()	returns the process identifier for the calling process
MPI_Bcast(message, root)	broadcasts "message" from a process with identifier "root" to all of the processes
MPI_Allreduce(A, B, MPI_SUM)	sums all the local copies of "A" in all the processes (reduction operation) and places the result in "B" on *all* of the processes (broadcast operation)
MPI_Wtime()	returns the number of seconds since some fixed, arbitrary point of time in the past

Table 1: Conceptual syntax and functionality of MPI routines which are used in Figure 2. For the exact syntax and usage, see [Gropp *et al.*1996, Snir *et al.*1997].

```
1:
2:
3:  MSE = LargeNumber;
4:
5:  Select k initial cluster centroids {m_j}_{j=1}^k;
6:
7:
8:  do {
9:      OldMSE = MSE;
10:     MSE' = 0;
11:     for j = 1 to k
12:         m'_j = 0; n'_j = 0;
13:     endfor;
14:     for i = 1 to n
15:         for j = 1 to k
16:             compute squared Euclidean
                    distance d^2(X_i, m_j);
17:         endfor;
18:         find the closest centroid m_ℓ to X_i;
19:         m'_ℓ = m'_ℓ + X_i; n'_ℓ = n'_ℓ + 1;
20:         MSE' = MSE' + d^2(X_i, m_ℓ);
21:     endfor;
22:     for j = 1 to k
23:
24:
25:         n_j = max(n'_j, 1); m_j = m'_j/n_j;
26:     endfor;
27:     MSE = MSE';
28: } while (MSE < OldMSE)
```

Figure 1: Sequential k-means Algorithm.

```
1:  P = MPI_Comm_size ();
2:  μ = MPI_Comm_rank ();
3:  MSE = LargeNumber;
4:  if (μ = 0)
5:      Select k initial cluster centroids {m_j}_{j=1}^k;
6:  endif;
7:  MPI_Bcast ({m_j}_{j=1}^k, 0);
8:  do {
9:      OldMSE = MSE;
10:     MSE' = 0;
11:     for j = 1 to k
12:         m'_j = 0; n'_j = 0;
13:     endfor;
14:     for i = μ * (n/P) + 1 to (μ + 1) * (n/P)
15:         for j = 1 to k
16:             compute squared Euclidean
                    distance d^2(X_i, m_j);
17:         endfor;
18:         find the closest centroid m_ℓ to X_i;
19:         m'_ℓ = m'_ℓ + X_i; n'_ℓ = n'_ℓ + 1;
20:         MSE' = MSE' + d^2(X_i, m_ℓ);
21:     endfor;
22:     for j = 1 to k
23:         MPI_Allreduce (n'_j, n_j, MPI_SUM);
24:         MPI_Allreduce (m'_j, m_j, MPI_SUM);
25:         n_j = max(n_j, 1); m_j = m_j/n_j;
26:     endfor;
27:     MPI_Allreduce (MSE', MSE, MPI_SUM);
28: } while (MSE < OldMSE)
```

Figure 2: Parallel k-means Algorithm. See Table 1 for a glossary of various MPI routines used above.

We begin by analyzing, in detail, the computational complexity of the sequential implementation of the k-means algorithm in Figure 1. We count each addition, multiplication, or comparison as one floating point operation (flop). It follows from Figure 1 that the amount of computation within each k-means iteration is constant, where each iteration consists of "distance calculations" in lines 14-21 and a "centroid recalculations" in lines 22-26. A careful examination reveals that the "distance calculations" require roughly $(3nkd+nk+nd)$ flops per iteration, where $3nkd$, nk, and nd correspond to lines 15-17, line 18, and line 19 in Figure 1, respectively. Also, "centroid recalculations" require approximately kd flops per iteration. Putting these together, we can estimate the computation complexity of the sequential implementation of the k-means algorithm as

$$(3nkd + nk + nd + kd) \cdot \mathfrak{I} \cdot T^{\text{flop}}, \tag{2}$$

where $\mathfrak{I}$ denotes the number of k-means iterations and T^{flop} denotes the time (in seconds) for a floating point operation. In this paper, we are interested in the case when the number of data points n is quite large in an absolute sense, and also large relative to d and k. Under this condition the serial complexity of the k-means algorithm is dominated by

$$T_1 \sim (3nkd) \cdot \mathfrak{I} \cdot T^{\text{flop}}. \tag{3}$$

By implementing a version of k-means on a distributed memory machine with P processors, we hope to reduce the total computation time by nearly a factor of P. Observe that the "distance calculations" in lines 14-21 of Figure 1 are inherently data parallel, that is, in principle, they can be executed asynchronously and in parallel for each data point. Furthermore, observe that these lines dominate the computational complexity in (2) and (3), when the number of data points n is large. In this context, a simple, but effective, parallelization strategy is to divide the n data points into P blocks (each of size roughly n/P) and compute lines 14-21 for each of these blocks in parallel on a different processor. This is the approach adopted in Figure 2.

For simplicity, assume that P divides n. In Figure 2, for $\mu = 0, 1, \cdots, P-1$, we assume that the process identified by "μ" has access to the data subset $\{X_i, i = (\mu) * (n/P) + 1, \cdots, (\mu+1) * (n/P)\}$. Observe that each of the P processes can carry out the "distance calculations" in parallel or asynchronously, if the centroids $\{m_j\}_{j=1}^k$ are available to each process. To enable this potential parallelism, in Figure 1, a local copy of the centroids $\{m_j\}_{j=1}^k$ is maintained for each process, see, line 7 and lines 22-26 in Figure 2 (see Table 1 for a glossary of the MPI calls used). Under this parallelization strategy, each process needs to handle only n/P data points, and hence we expect the total computation time for the parallel k-means to decrease to

$$T_P^{\text{comp}} = \frac{T_1}{P} \sim \frac{(3nkd) \cdot \mathfrak{I} \cdot T^{\text{flop}}}{P}. \tag{4}$$

In other words, as a benefit of parallelization, we expect the computational burden to be shared equally by all the P processors. However, there is also a price attached to this benefit, namely, the associated communication cost, which we now examine.

Before each new iteration of k-means can begin, all the P processes must communicate to recompute the centroids $\{m_j\}_{j=1}^k$. This global communication (and hence synchronization) is represented by lines 22-26 of Figure 2. Since, in each iteration, we must "MPI_Allreduce" roughly $d \cdot k$ floating point numbers, we can estimate the communication time for the parallel k-means to be

$$T_P^{\text{comm}} \sim d \cdot k \cdot \mathfrak{I} \cdot T_P^{\text{reduce}}, \tag{5}$$

where T_P^{reduce} denotes the time (in seconds) required to "MPI_Allreduce" a floating point number on P processors. On most architectures, one may assume that $T_P^{\text{reduce}} = O(\log P)$ [Culler *et al.*1996].

Line 27 in Figure 2 ensures that each of the P processes has a local copy of the total mean-squared-error "MSE", hence each process can independently decide on the convergence condition, that is, when to exit the "do{ $\cdots$ }while" loop.

In conclusion, each iteration of our parallel k-means algorithm consists of an asynchronous computation phase followed by a synchronous communication phase. The reader may compare Figures 1 and 2 line-by-line to see the precise correspondence of the proposed parallel algorithm with the serial algorithm. We stress that Figure 2 is optimized for understanding, and not for speed! In particular, in our actual implementation, we do not use $(2k+1)$ different "MPI_Allreduce" operations as suggested by lines 23, 24, and 27, but rather use a single block "MPI_Allreduce" by assigning a single, contiguous block of memory for the variables $\{m_j\}_{j=1}^k$, $\{n_j\}_{j=1}^k$, and MSE and a single, contiguous block of memory for the variables $\{m'_j\}_{j=1}^k$, $\{n'_j\}_{j=1}^k$, and MSE$'$.

We can now combine (4) and (5) to estimate the computational complexity of the parallel k-means algorithm as

$$T_P = T_P^{\text{comp}} + T_P^{\text{comm}} \sim \frac{(3nkd) \cdot \mathfrak{I} \cdot T^{\text{flop}}}{P} + d \cdot k \cdot \mathfrak{I} \cdot T_P^{\text{reduce}}. \tag{6}$$

It can be seen from (4) and (5) that the relative cost for the communication phase T_P^{comm} is insignificant compared to that for the computation phase T_P^{comp}, if

$$\boxed{\frac{P \cdot T_P^{\text{reduce}}}{3 \cdot T^{\text{flop}}} \ll n}. \tag{7}$$

Since the left-hand side of the above condition is a machine constant, as the number of data points n increases, we expect the relative cost for the communication phase compared to the computation phase to progressively decrease.

In the next section, we empirically study the performance of the proposed parallel k-means algorithm.

4 Performance and Scalability Analysis

Sequential algorithms are tested for correctness by seeing whether they give the right answer. For parallel programs, the right answer is not enough: we would like to decrease the execution time by adding more processors or we would like to handle larger data sets by using more processors. These desirable characteristics of a parallel algorithm are measured using "speedup" and "scaleup," respectively; we now empirically study these characteristics for the proposed parallel k-means algorithm.

4.1 Experimental Setup

We ran all of our experiments on an IBM SP2 with a maximum of 16 nodes. Each node in the multiprocessor is a Thin Node 2 consisting of a IBM POWER2 processor running at 160 MHz with 256 megabytes of main memory. The processors all run AIX level 4.2.1 and communicate with each other through the High-Performance Switch with HPS-2 adapters. The entire system runs PSSP 2.3 (Parallel System Support Program). See [Snir *et al.*1995] for further information about the SP2 architecture.

Our implementation is in C and MPI. All the timing measurements are done using the routine "MPI_Wtime()" described in Table 1. Our timing measurements ignore the I/O times (specifically, we ignore the time required to read in the data set from disk), since, in this paper, we are only interested in studying the efficacy of our parallel k-means algorithm. All the timing measurements were taken on an otherwise idle system. To smooth out any fluctuations, each measurement was repeated five times and each reported data point is to be interpreted as an average over five measurements.

For a given number of data points n and number of dimensions d, we generated a test data set with 8 clusters using the algorithm in [Milligan1985]. A public domain implementation of this algorithm "clusgen.c" is available from Dave Dubin (http://alexia.lis.uiuc.edu/dubin/). The advantage of such data generation is that we can generate as many data sets as desired with precisely specifiable characteristics.

As mentioned in Section 2, each run of the k-means algorithm depends on the choice of the starting cluster centroids. Specifically, the initial choice determines the specific local minimum of (1) that will be found by the algorithm, and it determines the number of k-means iterations. To eliminate the impact of the initial choice on our timing measurements, for a fixed data set, identical starting cluster centroids are used–irrespective of the number of processors used.

We are now ready to describe our experimental results.

4.2 Speedup

Relative speedup is defined as the ratio of the execution time for clustering a data set into k clusters on 1 processor to the execution time for identically clustering the same data set on P processors. Speedup is a summary of the efficiency of the parallel algorithm.

Using (3) and (6), we may write relative speedup of the parallel k-means roughly as

$$\text{Speedup} = \frac{(3nkd) \cdot \Im \cdot T^{\text{flop}}}{(3nkd) \cdot \Im \cdot T^{\text{flop}}/P + d \cdot k \cdot \Im \cdot T_P^{\text{reduce}}}, \tag{8}$$

which approaches the linear speedup of P when condition (7) is satisfied, that is, the number of data points n is large. We report three sets of experiments, where we vary n, d, and k, respectively.

Varying n: First, we study the speedup behavior when the number of data points n is varied. Specifically, we consider five data sets with $n = 2^{13}, 2^{15}, 2^{17}, 2^{19}$, and 2^{21}. We fixed the number of dimensions $d = 8$ and the number of desired clusters $k = 8$. We clustered each data set on $P = 1, 2, 4, 8$, and 16 processors. We report the measured execution times in Figure 3, and the corresponding relative speedup results in Figure 4, from where we can observe the following facts:

- For the largest data set, that is, $n = 2^{21}$, we observe a relative speedup of 15.62 on 16 processors. Thus, for large number of data points n our parallel k-means algorithm has nearly linear relative speedup.

  But, in contrast, for the smallest data set, that is, $n = 2^{11}$, we observe that relative speedup flattens at 6.22 on 16 processors.

- For a fixed number of processors, say, $P = 16$, as the number of data points increase from $n = 2^{11}$ to $n = 2^{21}$ the observed relative speedup generally increases from 6.22 to 15.62, respectively. In other words, our parallel k-means has an excellent *sizeup* behavior in the number of data points.

All these empirical facts are consistent with the theoretical analysis presented in the previous section; in particular, see condition (7).

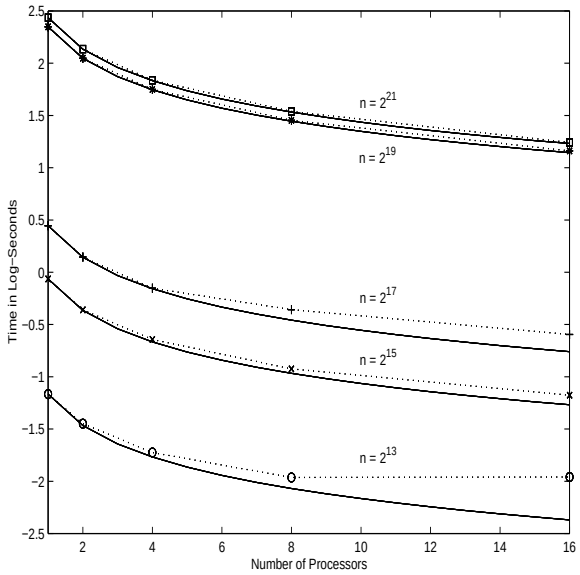


Figure 3: Speedup curves. We plot execution time in $\log_{10}$-seconds versus the number of processors. Five data sets are used with number of data points $n = 2^{13}, 2^{15}, 2^{17}, 2^{19}$, and 2^{21}. The number of dimensions $d = 8$ and the number of clusters $k = 8$ are fixed for all the five data sets. For each data set, the k-means algorithm required $\Im = 3, 10, 8, 164$ and 50 number of iterations, respectively. For each data set, a dotted line connects the observed execution times, while a solid line represents the "ideal" execution times obtained by dividing the observed execution time for 1 processor by the number of processors.

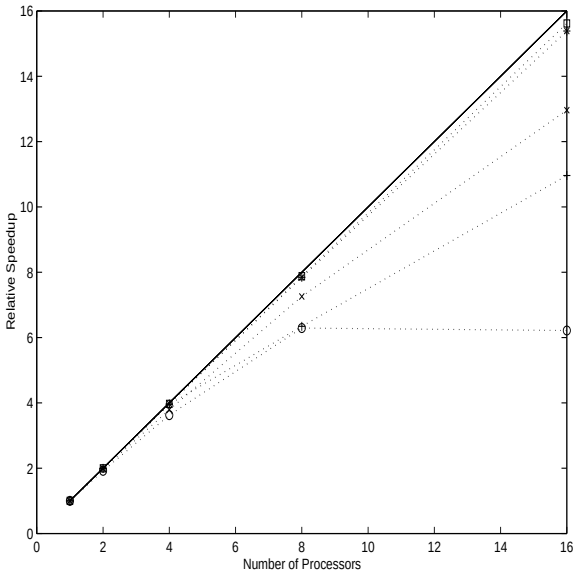


Figure 4: Relative Speedup curves for five data sets with points $n = 2^{13}, 2^{15}, 2^{17}, 2^{19}$, and 2^{21}. The number of dimensions $d = 8$ and the number of clusters $k = 8$ are fixed for all the five data sets. For each data set, the k-means algorithm required $\Im = 3, 10, 8, 164$ and 50 number of iterations, respectively. The solid line represents "ideal" linear relative speedup. For each data set, a dotted line connects observed relative speedups.

Varying d: Second, we study the speedup behavior when the number of dimensions d is varied. Specifically, we consider three data sets with $d = 2, 4$, and 8. We fixed the number of data points $n = 2^{21}$ and the number of desired clusters $k = 8$. We clustered each data set on $P = 1, 2, 4, 8$, and 16 processors. We report the measured relative speedup results in Figure 5.

Varying k: Finally, we study the speedup behavior when the number of desired clusters k is varied. Specifically, we clustered a fixed data set into $k = 2, 4, 8$, and 16 clusters. We fixed the number of data points $n = 2^{21}$ and the number of dimensions $d = 8$. We clustered the data set on $P = 1, 2, 4, 8$, and 16 processors. The corresponding relative speedup results are given in Figure 6.

In Figure 4, we observe nearly linear speedups between 15.42 to 15.53 on 16 processors. Similarly, in Figure 5, we observe nearly linear speedups between 15.08 to 15.65 on 16 processors. The excellent speedup numbers can be attributed to the fact that for $n = 2^{21}$ the condition (7) is satisfied. Also, observe that all the relative speedup numbers in Figures 5 and 6 are essentially independent of d and k, respectively. This is consistent with the fact that neither d nor k appears in the condition (7).

4.3 Scaleup

For a fixed data set (or a problem size), speedup captures the decrease in execution speed that can be obtained by increasing the number of processors. Another figure of merit of a parallel algorithm is *scaleup* which captures how well the parallel algorithm handles larger data sets when more processors are available. Our scaleup study measures execution times by keeping the problem size per processor fixed while increasing the number of processors. Since, we can increase the problem size in either the number of data points n, the number of dimensions d, or the number of desired clusters k, we can study scaleup with respect to each of these parameters at a time.

Relative scaleup of the parallel k-means algorithm with respect to n is defined as the ratio of the execution time (per iteration) for clustering a data set with n data points on 1 processor to the the execution time (per iteration) for clustering a data set with $n \cdot P$ data points on P processors–where the number of dimensions d and the number of desired clusters k are held constant. Observe that we measure execution time per iteration,

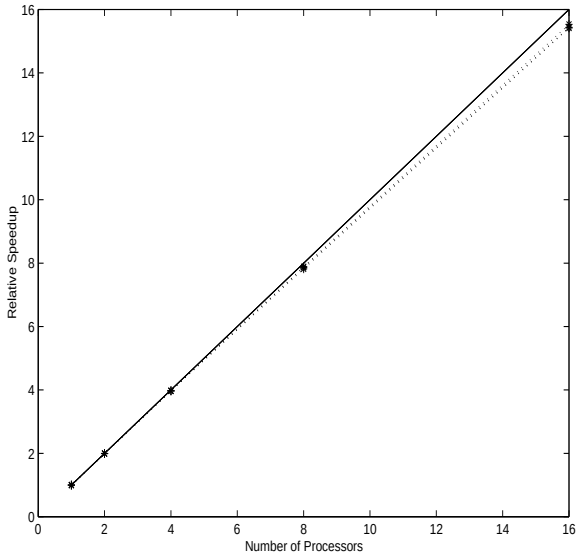


Figure 5: Relative speedup curves for three data sets with $d = 2, 4$, and 8. The number of data points $n = 2^{21}$ and the number of clusters $k = 8$ are fixed for all the three data sets. The solid line represents "ideal" linear relative speedup. For each data set, a dotted line connects observed relative speedups. It can be seen that relative speedups for different data sets are virtually indistinguishable from each other.

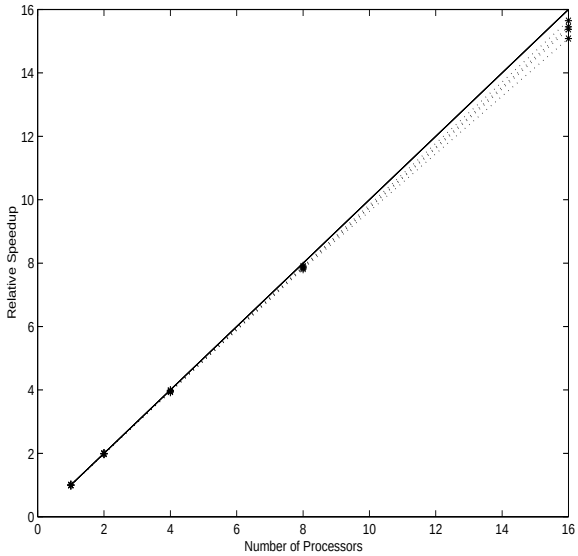


Figure 6: Relative speedup curves for four data sets with $k = 2, 4, 8$, and 16. The number of data points $n = 2^{21}$ and the number of dimensions $d = 8$ are fixed for all the four data sets. The solid line represents "ideal" linear relative speedup. For each data set, a dotted line connects observed relative speedups. It can be seen that relative speedups for different data sets are virtually indistinguishable from each other.

and not raw execution time. This is necessary since the k-means algorithm may require a different number of iterations $\mathfrak{I}$ for a different data set. Using (3) and (6), we can analytically write relative scaleup with respect to n as

$$\text{Scaleup} = \frac{(3nkd) \cdot T^{\text{flop}}}{(3nPkd) \cdot T^{\text{flop}}/P + d \cdot k \cdot T_P^{\text{reduce}}}. \quad (9)$$

It follows from (9) that if

$$\boxed{\frac{T_P^{\text{reduce}}}{3 \cdot T^{\text{flop}}} \ll n}, \quad (10)$$

then we expect relative scaleup to approach the constant 1. Observe that condition (10) is weaker than (7), and will be more easily satisfied for large number of data points n which is the case we are interested in. Relative scaleup with respect to either k or d can be defined analogously; we omit the precise definitions for brevity. The following experimental study shows that our implementation of parallel k-means has linear scaleup in n and k, and surprisingly better than linear scaleup in d.

Scaling n: To empirically study scaleup with respect to n, we clustered data sets with $n = 2^{21} \cdot P$ on $P = 1, 2, 4, 8, 16$ processors, respectively. We fixed the number of dimensions $d = 8$ and the number of desired clusters $k = 8$. The execution times per iteration are reported in Figure 7, from where it can be seen that the parallel k-means delivers virtually constant execution times in number of processors, and hence has excellent scaleup with respect to n. The largest data set with $n = 2^{21} \cdot 16 = 2^{25}$ is roughly 2 gigabytes. For this data set, our algorithm drives the SP2 at nearly 1.2 gigaflops. Observe that the main memory available on each of the 16 nodes is 256 megabytes, and hence this data set will not fit in the main memory of any single node, but easily fits in the combined main memory of 16 nodes. This is yet another benefit of parallelism–the ability to cluster significantly large data sets *in-core*, that is, in main memory.

Scaling k: To empirically study scaleup with respect to k, we clustered a data set into $k = 8 \cdot P$ clusters on $P = 1, 2, 4, 8, 16$ processors, respectively. We fixed the number of data points $n = 2^{21}$, and the number of dimensions $d = 8$. The execution times per iteration are reported in Figure 7, from where it can be seen that our parallel k-means delivers virtually constant execution times in number of processors, and hence has excellent scaleup with respect to k.

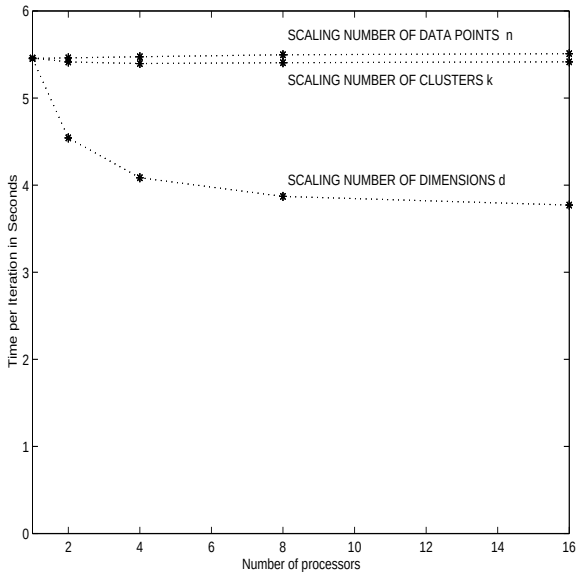


Figure 7: Scaleup curves. We plot execution time per iteration in seconds versus the number of processors. The same data set with $n = 2^{21}$, $d = 8$, and $k = 8$ is used for all the three curves–when the number of processors is equal to 1. For the "n" curve, the number of data points is scaled by the number of processors, while d and k are held constant. For the "k" curve, the number of clusters is scaled by the number of processors, while n and d are held constant. For the "d" curve, the number of dimensions is scaled by the number of processors, while n and k are held constant.

Scaling d: To empirically study scaleup with respect to d, we clustered data sets with the number of dimensions $d = 8 \cdot P$ on $P = 1, 2, 4, 8, 16$ processors, respectively. We fixed the number of data points $n = 2^{21}$, and the number of desired clusters $k = 8$. The execution times per iteration are reported in Figure 7, from where it can be seen that our parallel k-means delivers better than constant execution times in number of processors, and hence has surprisingly nice scaleup with respect to d. We conjecture that this phenomenon occurs due to the reduced loop overhead in the "distance calculations" as d increases (see Figure 2). The largest data set with $d = 8 \cdot 16 = 128$ is roughly 2 gigabytes. For this data set, our algorithm drives the SP2 at nearly 1.8 gigaflops.

5 Future Work

In this paper, we proposed a parallel k-means algorithm for distributed memory multiprocessors. Our algorithm is also easily adapted to shared memory multiprocessors where all processors have access to the same memory space. Many such machines are now currently available from a number of vendors. The basic strategy in adapting our algorithm to shared memory machine with P processors would be the same as that in this paper, namely, divide the set of data points n into P blocks (each of size roughly n/P) and compute distance calculations in lines 14-21 of Figure 1 for each of these blocks in parallel on a different processor while ensuring that each processor has access to a separate copy of the centroids $\{m_j\}_{j=1}^{k}$. Such an algorithm can be implemented on a shared memory machine using threads [Northrup1996].

It is well known that the k-means algorithm is a hard thresholded version of the expectation-maximization (EM) algorithm [McLachlan and Krishnan1996]. We believe that the EM algorithm can be effectively parallelized using essentially the same strategy as that used in this paper.

References

[Agrawal and Shafer1996] R. Agrawal and J. C. Shafer. Parallel mining of association rules: Design, implementation, and experience. *IEEE Trans. Knowledge and Data Eng.*, 8(6):962–969, 1996.

[Alsabti *et al.*1998] K. Alsabti, S. Ranka, and V. Singh. An efficient k-means clustering algorithm. In *Proceedings of IPPS/SPDP Workshop on High Performance Data Mining*, 1998.

[Bottou and Bengio1995] L. Bottou and Y. Bengio. Convergence properties of the k-means algorithms. In G. Tesauro and D. Touretzky, editors, *Advances in Neural Information Processing Systems 7*, pages 585–592. The MIT Press, Cambridge, MA, 1995.

[Chattratichat *et al.*1997] J. Chattratichat, J. Darlington, M. Ghanem, Y. Guo, H. Hüning, M. Köhler, J. Sutiwaraphun, H. W. To, and D. Yang. Large scale data mining: Challenges and responses. In D. Pregibon and R. Uthurusamy, editors, *Proceedings Third International Conference on Knowledge Discovery and Data Mining, Newport Beach, CA*, pages 61–64. AAAI Press, 1997.

[Cheung and Xiao1999] D. W. Cheung and Y. Xiao. Effect of data distribution in parallel mining of associations. *Data Mining and Knowledge Discovery*, 1999. to appear.

[Culler *et al.*1996] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, 1996.

[Cutting *et al.*1992] D. R. Cutting, D. R. Karger, J. O. Pedersen, and J. W. Tukey. Scatter/gather: A

cluster-based approach to browsing large document collections. In *ACM SIGIR*, 1992.

[Dhillon *et al.*1998] I. S. Dhillon, D. S. Modha, and W. S. Spangler. Visualizing class structure of multidimensional data. In S. Weisberg, editor, *Proceedings of the 30th Symposium on the Interface: Computing Science and Statistics, Minneapolis, MN*, May 13–16 1998.

[Fayyad *et al.*1996] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, 1996.

[Freitas and Lavington1998] A. A. Freitas and S. H. Lavington. *Mining Very Large Databases with Parallel Processing*. Kluwer Academic Publishers, 1998.

[Fukunaga and Narendra1975] K. Fukunaga and P. M. Narendra. A branch and bound algorithm for computing k-nearest neighbors. *IEEE Trans. Comput.*, pages 750–753, 1975.

[Garey *et al.*1982] M. R. Garey, D. S. Johnson, and H. S Witsenhausen. The complexity of the generalized Lloyd-Max problem. *IEEE Trans. Inform. Theory*, 28(2):255–256, 1982.

[Gersho and Gray1992] A. Gersho and R. M. Gray. *Vector quantization and signal compression*. Kluwer Academic Publishers, 1992.

[Gropp *et al.*1996] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. The MIT Press, Cambridge, MA, 1996.

[Han *et al.*1997] E.-H. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. In *SIGMOD Record: Proceedings of the 1997 ACM-SIGMOD Conference on Management of Data, Tucson, AZ, USA*, pages 277–288, 1997.

[Hartigan1975] J. A. Hartigan. *Clustering Algorithms*. Wiley, 1975.

[Joshi *et al.*1998] M. V. Joshi, G. Karypis, and V. Kumar. ScalParC: A new scalable and efficient parallel classification algorithm for mining large datasets. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing, Orlando, FL, USA*, pages 573–579, 1998.

[Kargupta *et al.*1997] H. Kargupta, I. Hamzaoglu, B. Stafford, V. Hanagandi, and K. Buescher. PADMA: Parallel data mining agents for scalable text classification. In *Proceedings of the High Performance Computing, Atlanta, GA, USA*, pages 290–295, 1997.

[McLachlan and Krishnan1996] G. J. McLachlan and T. Krishnan. *The EM Algorithm and Extentions*. Wiley, 1996.

[Milligan1985] G. Milligan. An algorithm for creating artificial test clusters. *Psychometrika*, 50(1):123–127, 1985.

[Northrup1996] C. J. Northrup. *Programming with UNIX Threads*. John Wiley & Sons, 1996.

[Shafer *et al.*1996] J.C. Shafer, R. Agrawal, and M. Mehta. A scalable parallel classifier for data mining. In *Proc. 22nd International Conference on VLDB, Mumbai, India*, 1996.

[Shaw and King1992] C. T. Shaw and G. P. King. Using cluster analysis to classify time series. *Physica D*, 58:288–298, 1992.

[Smyth *et al.*1997] P. Smyth, M. Ghil, K. Ide, J. Roden, and A. Fraser. Detecting atmospheric regimes using cross-validated clustering. In D. Pregibon and R. Uthurusamy, editors, *Proceedings Third International Conference on Knowledge Discovery and Data Mining, Newport Beach, CA*, pages 61–64. AAAI Press, 1997.

[Snir *et al.*1995] M. Snir, P. Hochschild, D. D. Frye, and K. J. Gildea. The communication software and parallel environment of the IBM SP2. *IBM Systems Journal*, 34(2):205–221, 1995.

[Snir *et al.*1997] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, Cambridge, MA, 1997.

[Srivastava *et al.*1998]
A. Srivastava, E.-H. Han, V. Kumar, and V. Singh. Parallel formulations of decision-tree classification algorithms. In *Proc. 1998 International Conference on Parallel Processing*, 1998.

[Stolorz and Musick1997] Paul Stolorz and Ron Musick. *Scalable High Performance Computing for Knowledge Discovery and Data Mining*. Kluwer Academic Publishers, 1997.

[Zaki *et al.*1997] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. New parallel algorithms for fast discovery of association rule. *Data Mining and Knowledge Discovery*, 1(4):343–373, 1997.

[Zaki *et al.*1998] M. J. Zaki, C. T. Ho, and R. Agrawal. Parallel classification for data mining on shared-memory multiprocessors. Technical report, IBM Almaden Research Center, 1998.

[Zhang *et al.*1996] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: An efficient data clustering method for very large databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data, Montreal, Canada*, 1996.

Parallel Sequence Mining on Shared-Memory Machines

Mohammed J. Zaki
Computer Science Dept., Rensselaer Polytechnic Institute, Troy, NY 12180

Abstract

We present pSPADE, a parallel algorithm for fast discovery of frequent sequences in large databases. pSPADE decomposes the original search space into smaller suffix-based classes. Each class can be solved in main-memory using efficient search techniques, and simple join operations. Further each class can be solved independently on each processor requiring no synchronization. However, dynamic inter-class and intra-class load balancing must be exploited to ensure that each processor gets an equal amount of work. Experiments on a 12 processor SGI Origin 2000 shared memory system show good speedup and scaleup results.

1 Introduction

The sequence mining task is to discover a sequence of attributes (items), shared across time among a large number of objects (transactions) in a given database. The task of discovering all frequent sequences in large databases is quite challenging. The search space is extremely large. For example, with m attributes there are $O(m^k)$ potentially frequent sequences of length at most k. Clearly, sequential algorithms cannot provide scalability, in terms of the data size and the performance, for large databases. We must therefore rely on parallel multiprocessor systems to fill this role.

Previous work on parallel sequence mining has only looked at distributed-memory machines [Shintani and Kitsuregawa, 1998]. In this paper we look at shared-memory systems, which are capable of delivering high performance for low to medium degree of parallelism at an economically attractive price. SMP machines are the dominant types of parallel machines currently used in industry. Individual nodes of parallel distributed-memory machines are also increasingly being designed to be SMP nodes.

Our platform is a 12 processor SGI Origin 2000 system, which is a cache-coherent non-uniform memory access (CC-NUMA) machine. For cache coherence the hardware ensures that locally cached data always reflects the latest modification by any processor. It is NUMA because reads/writes to local memory are cheaper than reads/writes to a remote processor's memory. The main challenge in obtaining high performance on these systems is to ensure good *data locality*, making sure that most read/writes are to local memory, and reducing/eliminating *false sharing*, which occurs when two different shared variables are (coincidentally) located in the same cache block, causing the block to be exchanged between the processors due to coherence maintenance operations, even though the processors are accessing different variables. Of course, the other factor influencing parallel performance for any system is to ensure good *load balance*, i.e., making sure that each processor gets an equal amount of work.

In this paper we present pSPADE, a parallel algorithm for discovering the set of all frequent sequences, targeting shared-memory systems, the first such study. pSPADE has been designed such that it has very good locality and has virtually no false sharing. Further, pSPADE is an asynchronous algorithm, in that it requires no synchronization among processors, except when a load imbalance is detected. We carefully consider several design alternatives in terms of data and task parallelism, and in terms of the load balancing strategy used, before adopting the best approach in pSPADE. An extensive set of experiments is performed on the SGI Origin 2000 machine. pSPADE delivers good speedup and scales linearly in the database size.

The rest of the paper is organized as follows: We describe the sequence discovery problem in Section 2. Section 3 describes the serial algorithm, while the design and implementation issues for pSPADE are presented in Section 4. An experimental study is presented in Section 5, and we conclude in Section 6.

2 Sequence Mining

The problem of mining sequential patterns can be stated as follows: Let $\mathcal{I} = \{i_1, i_2, \cdots, i_m\}$ be a set of m

distinct attributes, also called *items*. An *itemset* is a non-empty unordered collection of items (without loss of generality, we assume that items of an itemset are sorted in increasing order). A *sequence* is an ordered list of itemsets. An itemset i is denoted as $(i_1 i_2 \cdots i_k)$, where i_j is an item. An itemset with k items is called a *k-itemset*. A sequence α is denoted as $(\alpha_1 \mapsto \alpha_2 \mapsto \cdots \mapsto \alpha_q)$, where the sequence *element* α_j is an itemset. A sequence with k items ($k = \sum_j |\alpha_j|$) is called a *k-sequence*. For example, $(B \mapsto AC)$ is a 3-sequence. An item can occur only once in an itemset, but it can occur multiple times in different itemsets of a sequence. A sequence $\alpha = (\alpha_1 \mapsto \alpha_2 \mapsto \cdots \mapsto \alpha_n)$ is a *subsequence* of another sequence $\beta = (\beta_1 \mapsto \beta_2 \mapsto \cdots \mapsto \beta_m)$, denoted as $\alpha \preceq \beta$, if there exist integers $i_1 < i_2 < \cdots < i_n$ such that $a_j \subseteq b_{i_j}$ for all a_j. For example the sequence $(B \mapsto AC)$ is a subsequence of $(AB \mapsto E \mapsto ACD)$, since the sequence elements $B \subseteq AB$, and $AC \subseteq ACD$. On the other hand the sequence $(AB \mapsto E)$ is not a subsequence of (ABE), and vice versa. We say that α is a proper subsequence of β, denoted $\alpha \prec \beta$, if $\alpha \preceq \beta$ and $\beta \not\preceq \alpha$. A sequence is *maximal* if it is not a subsequence of any other sequence.

DATABASE

CID	TID	Items
1	10	A B
	20	B
	30	A B
2	20	A C
	30	A B C
	50	B
3	10	A
	30	B
	40	A
4	30	A B
	40	A
	50	B

FREQUENT SET (75% Minimum Support)
{A, A->A, B->A, B, AB, A->B, B->B, AB->B}

Figure 1: Original Database

A *transaction* $\mathcal{T}$ has a unique identifier and *contains* a set of items, i.e., $\mathcal{T} \subseteq \mathcal{I}$. A *customer*, $\mathcal{C}$, has a unique identifier and has associated with it a list of transactions $\{\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_n\}$. Without loss of generality, we assume that no customer has more than one transaction with the same time-stamp, so that we can use the transaction-time as the transaction identifier. We also assume that the list of customer transactions is sorted by the transaction-time. Thus the list of transactions of a customer is itself a sequence $\mathcal{T}_1 \mapsto \mathcal{T}_2 \mapsto \cdots \mapsto \mathcal{T}_n$, called the *customer-sequence*. The database, $\mathcal{D}$, consists of a number of such customer-sequences. A customer-sequence, $\mathcal{C}$, is said to *contain* a sequence α, if $\alpha \preceq \mathcal{C}$, i.e., if α is a subsequence of the customer-sequence $\mathcal{C}$. The *support* or *frequency* of a sequence, denoted $\sigma(\alpha)$, is the the total number of customers that contain this sequence. Given a user-specified threshold called the *minimum support* (denoted *min_sup*), we say that a sequence is *frequent* if occurs more than *min_sup* times. The set of frequent k-sequences is denoted as $\mathcal{F}_k$. Given a database $\mathcal{D}$ of customer sequences and *min_sup*, the problem of mining sequential patterns is to find all frequent sequences in the database. For example, consider the customer database shown in figure 1. The database has three items (A, B, C), four customers, and twelve transactions in all. The figure also shows all the frequent sequences with a minimum support of 75% or 3 customers.

3 The Serial SPADE Algorithm

Several sequence mining algorithms have been proposed in recent years [Srikant and Agrawal, 1996, Mannila *et al.*, 1997, Oates *et al.*, 1997]. GSP [Srikant and Agrawal, 1996] is one of the best known serial algorithms. Recently, SPADE [Zaki, 1998] was shown to outperform GSP by a factor of two in the general case, and by a factor of ten with a pre-processing step. In this section we describe SPADE [Zaki, 1998], a serial algorithm for fast discovery of frequent sequences, which forms the basis for the parallel pSPADE algorithm.

Sequence Lattice SPADE uses the observation that the subsequence relation $\preceq$ defines a partial order on the set of sequences, also called a *specialization relation*. If $\alpha \preceq \beta$, we say that α is more general than β, or β is more specific than α. The second observation used is that the relation $\preceq$ is a *monotone specialization relation* with respect to the frequency $\sigma(\alpha, \mathcal{D})$, i.e., if β is a frequent sequence, then all subsequences $\alpha \preceq \beta$ are also frequent. The algorithm systematically searches the sequence lattice spanned by the subsequence relation, from the most general to the maximally specific frequent sequences in a breadth/depth-first manner. For example, Figure 2 A) shows the lattice of frequent sequences for our example database.

Support Counting Most of the current sequence mining algorithms [Srikant and Agrawal, 1996] assume a *horizontal* database layout such as the one shown in Figure 1. In the horizontal format the database consists of a set of customers (*cid's*). Each customer has a set of transactions (*tid's*), along with the items contained in the transaction. In contrast, we use a *vertical* database layout, where we associate with each item X in the sequence lattice its *idlist*, denoted $\mathcal{L}(X)$, which is a list of all customer (*cid*) and transaction identifiers (*tid*) pairs

containing the atom. The vertical idlist format, called *binary association table*, has also been used in [Holsheimer *et al.*, 1996] for parallel data mining. Figure 2 B) shows the idlists for all the items.

Given the sequence idlists, we can determine the support of any k-sequence by simply intersecting the idlists of any two of its $(k-1)$ length subsequences. In particular, we use the two $(k-1)$ length subsequences that share a common suffix (the generating sequences) to compute the support of a new k length sequence. A simple check on the cardinality of the resulting idlist tells us whether the new sequence is frequent or not. Figure 2 shows this process pictorially. It shows the initial vertical database with the idlist for each item. There are two kinds of intersections: *temporal* and *equality*. For example, Figure 2 shows the idlist for $A \rightarrow B$ obtained by performing a temporal intersection on the idlists of A and B, i.e., $\mathcal{L}(A \rightarrow B) = \mathcal{L}(A) \cap_t \mathcal{L}(B)$. This is done by looking if, within the same *cid*, A occurs before B, and listing all such occurrences. On the other hand the idlist for $AB \rightarrow B$ is obtained by an equality intersection, i.e., $\mathcal{L}(AB \rightarrow B) = \mathcal{L}(A \rightarrow B) \cap_e \mathcal{L}(B \rightarrow B)$. Here we check to see if the two subsequences occur within the same *cid* at the same time. Additional details can be found in [Zaki, 1998].

To use only a limited amount of main-memory SPADE breaks up the sequence search space into small, independent, manageable chunks which can be processed in memory. This is accomplished via suffix-based partition. We say that two k length sequences are in the same equivalence class or *partition* if they share a common $k-1$ length suffix. The partitions, such as $\{[A], [B]\}$, based on length 1 suffixes are called *parent partitions*. Each parent partition is independent in the sense that it has complete information for generating all frequent sequences that share the same suffix. For example, if a class $[X]$ has the elements $Y \rightarrow X$, and $Z \rightarrow X$. The possible frequent sequences at the next step are $Y \rightarrow Z \rightarrow X$, $Z \rightarrow Y \rightarrow X$, and $(YZ)\ rightarrow X$. No other item Q can lead to a frequent sequence with the suffix X, unless (QX) or $Q \rightarrow X$ is also in $[X]$.

SPADE recursively decomposes the sequences at each new level into even smaller independent classes, which produces a computation tree of independent classes as shown in Figure 2B). This computation tree is processed in a breadth-first manner within each parent class. Figure 3 shows the pseudo-code for SPADE; we refer the reader to [Zaki, 1998] for more details.

SPADE $(min\_sup, \mathcal{D})$:
  $\mathcal{C} = \{$ parent classes $C_i = [X_i]\}$;
  for each $C_i \in \mathcal{C}$ **do** *Enumerate-Frequent-Seq*(C_i);

//$PrevL$ is list of frequent classes from previous level
//$NewL$ is list of new frequent classes for current level
Enumerate-Frequent-Seq($PrevL$**):**
  for $(; PrevL \neq \emptyset; PrevL = PrevL.next())$
    $NewL = NewL \cup$ *Get-New-Classes* $(PrevL.item())$;
  if $(NewL \neq \emptyset)$ **then** *Enumerate-Frequent-Seq*$(NewL)$;

Get-New-Classes(S**):**
  for all atoms $A_i \in S$ **do**
    for all atoms $A_j \in S$, with $j > i$ **do**
      $R = A_i \cup A_j$;
      $\mathcal{L}(R) = \mathcal{L}(A_i) \cap \mathcal{L}(A_j)$;
      if $(\sigma(R) \geq min\_sup)$ **then** $C_i = C_i \cup \{R\}$;
    $CList = CList \cup C_i$;
  return $CList$;

Figure 3: SPADE pseudo-code

4 The Parallel pSPADE Algorithm

While parallel association mining has attracted wide attention [Agrawal and Shafer, 1996, Zaki *et al.*, 1997], there has been relatively less work on parallel mining of sequential patterns. Three distributed-memory parallel algorithms based on GSP were presented in [Shintani and Kitsuregawa, 1998]. pSPADE is the first algorithm for shared-memory systems.

pSPADE is best understood by visualizing the computation as a dynamically expanding irregular tree of independent suffix-based classes, as shown in Figure 4. This tree represents the search space for the algorithm. There are three independent suffix-based parent equivalence classes. These are the only classes visible at the beginning of computation. Since we have a shared-memory machine, there is only one copy on disk of the database in the vertical idlist format. It can be accessed by any processor, via a local file descriptor. Given that each class in the tree can be solved independently the crucial issue is how to achieve a good load balance, so that each processor gets an equal amount of work. We would also like to maximize locality and minimize/eliminate cache contention.

There are two main paradigms that may be utilized in the implementation of parallel sequence mining: a *data parallel* approach and a *task parallel* approach. In data parallelism P processors work on distinct portions of the database, but synchronously process the global computation tree. It essentially exploits intra-class parallelism, i.e., the parallelism available within a class. In task parallelism, the processors share the database, but work on different classes in parallel, asynchronously processing the computation tree. This scheme is thus

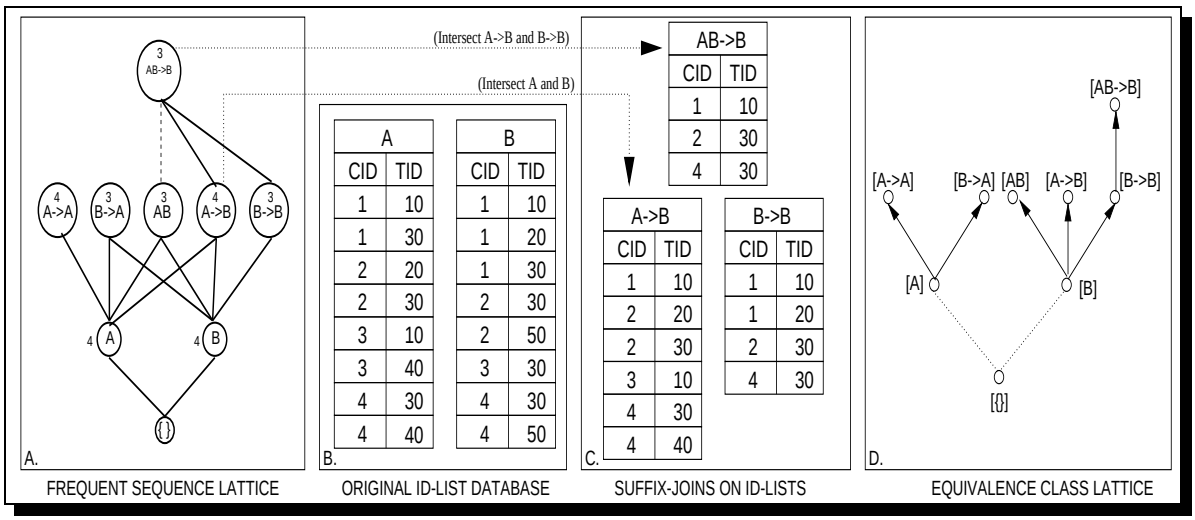


Figure 2: SPADE: Lattices and Joins

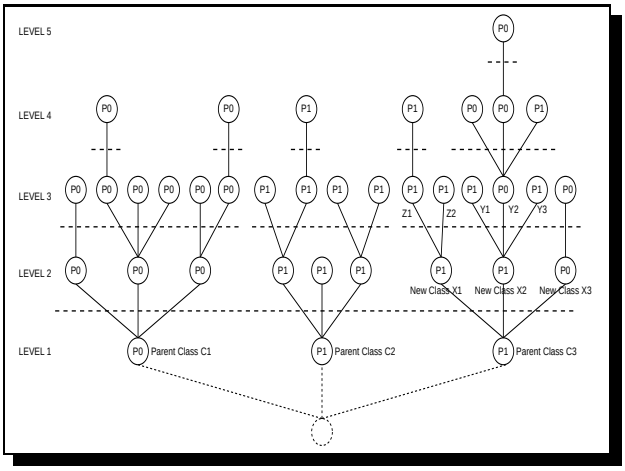


Figure 4: Dynamic, Irregular Computation Tree of Classes

based on inter-class parallelism.

4.0.1 Data Parallelism

For sequence mining, data parallelism can come in two flavors. The first case corresponds to *idlist parallelism*, in which we physically partition each idlist into P ranges over the customer sequence $cids$ (for example, processor 0 is responsible for the cid range $0 \cdots l$, processor 1 for range $l + 1 \cdots 2l$, and so on). Each processor is responsible for $1/P$ of the $cids$. The other case corresponds to *join parallelism*, where each processor picks a sequence and performs intersections with the other sequences in the same class, generating new classes for the next level.

Idlist Parallelism There are two ways of implementing the idlist parallelism. In the first method each intersection is performed in parallel among the P processors. Each processor performs the intersection over its cid range, and increments support in a shared variable. A barrier synchronization must be performed to make sure that all processors have finished their intersection for the candidate. Finally, based on the support this candidate may be discarded or added to the new class. This scheme suffers from massive synchronization overheads. As we shall see in Section 5 for some values of minimum support we performed around 0.6 million intersections. This scheme will require as many barrier synchronizations. The other method uses a level-wise approach. In other words, at each new level of the computation tree, each processor processes all the classes at that level, performing intersections for each candidate, but only over its local database portion. The local supports are stored in a local array to prevent false sharing among processors. After a barrier synchronization signals that all processors have finished processing the current level, a sum-reduction is performed in parallel to determine the global support of each candidate. The frequent sequences are then retained for the next level, and the same process is repeated for other levels until no more frequent sequences are found.

We implemented the level-wise idlist parallelism and found that it performed very poorly. In fact, we got a speed-down as we increased the number of processors (see Section 5). Even though we tried to minimize the synchronization as much as possible, performance was still unacceptable. Since a candidate's memory cannot be freed until the end of a level, the memory consumption of this approach is extremely high. We

were unable to run this algorithm for low values of minimum support. Also, when the local memory is not sufficient the Origin allocates remote memory for the intermediate idlists, causing a performance hit due to the NUMA architecture.

Join Parallelism In join parallelism each processor performs intersections for different sequences within the same class. Once the current class has been processed, the processors must synchronize before moving on to the next class. While we have not implemented this approach, we believe that it will fare no better than idlist parallelism. The reason is that it requires one synchronization per class, which is better than the single candidate idlist parallelism, but still much worse than the level-wise idlist parallelism, since there can be many classes.

4.0.2 Task Parallelism

In task parallelism all processors have access to the entire database, but they work on separate classes. We present a number of load balancing approaches starting with a static load balancing scheme and moving on to a more sophisticated dynamic load balancing strategy.

Static Load Balancing (SLB) Let $\mathcal{C} = \{C_1, C_2, C_3\}$ represent the set of the parent classes at level 1 as shown in Figure 4. We need to schedule the classes among the processors in a manner minimizing load imbalance. In our approach an entire class is scheduled on one processor. Load balancing is achieved by assigning a weight to each equivalence class based on the number of elements in the class. Since we have to consider all pairs of items for the next iteration, we assign the weight $W1_i = \binom{|C_i|}{2}$ to the class C_i. Once the weights are assigned we generate a schedule using a greedy heuristic. We sort the classes on the weights (in decreasing order), and assign each class in turn to the least loaded processor, i.e., one having the least total weight at that point. Ties are broken by selecting the processor with the smaller identifier. We also studied the effect of other heuristics for assigning class weights, such as $W2_i = \sum_j |\mathcal{L}(A_j)|$ for all items A_j in the class C_i. This cost function gives each class a weight proportional to the sum of the supports of all the items. We also tried a cost function that combines the above two, i.e., $W3_i = \binom{|C_i|}{2} \cdot \sum_j |\mathcal{L}(A_j)|$. We did not observe any significant benefit of one weight function over the other, and decided to use $W1$.

Figure 5 shows the pseudo-code for the SLB algorithm. We schedule the classes on different processors based on the class weights. Once the classes have been scheduled, the computation proceeds in a purely asynchronous manner since there is never any need to synchronize or share information among the processors. If we apply $W1$ to the class tree shown in Figure 4, we get $W1_1 = W1_2 = W1_3 = 3$. Using the greedy scheduling scheme on two processors, P_0 gets the classes C_1 and C_3, and P_1 gets the class C_2. We immediately see that SLB suffers from load imbalance, since after processing C_1, P_0 will be busy working on C_3, while after processing C_2, P_1 has no more work. The main problem with SLB is that, given the irregular nature of the computation tree there is no way of accurately determining the amount of work per class statically.

Inter-Class Dynamic Load Balancing (CDLB) To get better load balancing we utilize inter-class dynamic load balancing. Instead of a static or fixed class assignment of SLB, we would like each processor to dynamically pick a new class to work on from the list of classes not yet processed. We also make use of the class weights in the CDLB approach. First, we sort the parent classes in decreasing order of their weight. This forms a logical central task queue of independent classes. Each processor atomically grabs one class from this logical queue. It processes the class completely and then grabs the next available class. Note that each class usually has a non-trivial amount of work, so that we don't have to worry too much about contention among processors to acquire new tasks. Since classes are sorted on their weights, processors first work on large classes before tackling smaller ones, which helps to achieve a greater degree of load balance. The pseudo-code for CDLB algorithm appears in Figure 5. The *compare-and-swap* (CAS) is an atomic primitive on the Origin. It compares $classid$ with i. If they are equal it replaces $classid$ with $i+1$, returning a 1, else it returns a 0. CAS ensures that processors acquire separate classes.

If we apply CDLB to our example computation tree in Figure 4, we might expect a scenario as follows: In the beginning P_1 grabs C_1, and P_0 acquires C_2. Since C_2 has less work, P_0 will grab the next class C_3 and work on it. Then P_1 becomes free and finds that there is no more work, while P_0 is still busy. For this example, CDLB did not buy us anything over SLB. However, when we have a large number of parent classes CDLB has a clear advantage over SLB, since a processor grabs a new class only when it has processed its current class. This way only the free processors will acquire new classes, while others continue to process their current class, delivering good processor utilization. We shall see in Section 5 that CDLB can provide up to 40% improvement over SLB. We should reiterate that the processing of classes is still asynchronous. For both

```
SLB (min_sup, D):
  C = { parent classes C_i = [X_i]};
  Sort-on-Weight(C);
  for all C_i ∈ C do
    P_j = Proc-with-Min-Weight();
    S_{P_j} = S_{P_j} ∪ C_i;
  parallel for all C_i ∈ S_{P_j} do
    Enumerate-Frequent-Seq(C_i);
```

```
CDLB (min_sup, D):
  C = { parent classes C_i = [X_i]};
  Sort-on-Weight(C);

  shared int classid=0;
  parallel for (i = 0; i < |C|; i++)
    if (compare_and_swap (classid, i, i+1))
      Enumerate-Frequent-Seq(C_i);
```

Figure 5: The SLB (Static Load Balancing) and CDLB (Dynamic Load Balancing) Algorithms

SLB and CDLB, false sharing doesn't arise, and all work is performed on local memory, resulting in good locality.

Recursive Dynamic Load Balancing (RDLB) While CDLB improves over SLB by exploiting dynamic load balancing, it does so only at the inter-class level, which may be too coarse-grained to achieve a good workload balance. RDLB addresses this by exploiting both inter-class and intra-class parallelism. To see where the intra-class parallelism can be exploited, lets examine the behavior of CDLB. As long as there are more parent classes remaining, each processor acquires a new class and processes it completely. If there are no more parent classes left, the free processors are forced to idle. The worst case happens when $P-1$ processors are free and only one is busy, especially if the last class has a deep computation tree (although we try to prevent this case from happening by sorting the classes, so that the smaller ones are at the end, it can still happen). We can fix this problem if we can provide a mechanism for the free processors to join the busy ones. We accomplish this by recursively applying the CDLB strategy at each new level, but only if there is some free processor waiting for more work. Since each class is independent, we can treat each class at the new level in the same way we treated the parent classes, so that different processors can work on different classes that the new level.

Figure 6 shows the pseudo-code for the pSPADE algorithm, which uses a recursive dynamic load balancing (RDLB) scheme. We start with the parent classes and insert them in the global class list, $GlobalQ$. A processor atomically acquires classes from this list until all parent classes have been taken. At that point the processor increments $FreeCnt$, and waits for more work. When a processor is processing the classes at some level, it periodically checks if there is any free processor. If so, it inserts the remaining classes at that level ($PrevL$) in $GlobalQ$, emptying $PrevL$ in the process, and sets $GlobalFlg$. This processor then quits the loop on line 16, and continues working on the new classes ($NewL$)

```
1. shared int FreeCnt = 0; //Number of free processors
2. shared int GlobalFlg = 0; //Is there more work?
3. shared list GlobalQ; //Global list of classes

pSPADE (min_sup, D):
4.   GlobalQ = C = { parent classes C_i = [X_i]};
5.   Sort-on-Weight(C);
6.   Process-GlobalQ();
7.   FreeCnt++;
8.   while (FreeCnt ≠ P)
9.     if (GlobalFlg) then
10.      FreeCnt--; Process-GlobalQ(); FreeCnt++;
Process-GlobalQ():
11.   shared int classid = 0;
12.     parallel for (i = 0; i < GlobalQ.size(); i++)
13.       if (compare_and_swap (classid, i, i+1))
14.         RDLB-Enumerate-Frequent-Seq(C_i);
15.   GlobalFlg = 0;
RDLB-Enumerate-Frequent-Seq(PrevL):
16.   for (; PrevL ≠ ∅; PrevL = PrevL.next())
17.     if (FreeCnt > 0) then
18.       Add-to-GlobalQ(PrevL.next()); GlobalFlg = 1;
19.     NewL = NewL ∪ Get-New-Classes (PrevL.item());
20.   if (NewL ≠ ∅) then RDLB-Enumerate-Frequent-Seq(NewL);
```

Figure 6: The pSPADE Algorithm (using RDLB)

generated before a free processor was detected. When a waiting processor sees that there is more work, it starts working on the classes in $GlobalQ$. When there is no more work, $FreeCnt$ equals the number of processors P, and the computation stops.

Let's illustrate the above algorithm by looking at the computation tree in Figure 4. The nodes are marked by the processors that work on them. First, at the parent class level, P_0 acquires C_1, and P_1 acquires C_2. Since C_2 is smaller, P_1 grabs class C_3, and starts processing it. It generates three new classes at the next level, $NewL = \{X_1, X_2, X_3\}$, which becomes $PrevL$ when P_1 start the next level. Let's assume that P_1 finishes processing X_1, and inserts classes Z_1, Z_2 in the new $NewL$. In the meantime, P_0 becomes free. Before processing X_2, P_1 notices in line 17, that there is a free processor. At this point P_1 inserts X_3 in $GlobalQ$, and empties $PrevL$. It then continues to work on X_2, inserting Y_1, Y_2, Y_3 in $NewL$. P_0 sees the new insertion in $GlobalQ$ and start working on X_3 in its entirety. P_0 meanwhile starts processing the next

Dataset	C	T	S	I	D	Size	MinSup
C10T5S4I1.25D1M	10	5	4	1.25	1M	320MB	0.25%
C10T5S4I2.5D1M	10	5	4	2.5	1M	320MB	0.33%
C20T2.5S4I1.25D1M	20	2.5	4	1.25	1M	440MB	0.25%
C20T2.5S4I2.5D1M	20	2.5	4	2.5	1M	440MB	0.25%
C20T5S8I1.25D1M	20	5	8	1.25	1M	640MB	0.33%
C20T5S8I2D1M	20	5	8	2	1M	640MB	0.5%
C5T2.5S4I1.25DxM	5	2.5	4	1.25	1M-10M	110MB-1.1GB	0.25%-0.01%

Table 1: Synthetic Datasets

level classes, $\{Z_1, Z_2, Y_1, Y_2, Y_3\}$. If at any stage it detects a free processor, it will repeat the procedure described above recursively. Figure 4 shows a possible execution sequence for the class C_3. The RDLB scheme of pSPADE preserves the good features of CDLB, i.e., it dynamically schedules entire parent classes on separate processors, for which the work is purely local, requiring no synchronization. So far only inter-class parallelism has been exploited. Intra-class parallelism is required only for a few (hopefully) small classes towards the end of the computation. We simply treat these as new parent classes, and schedule each class on a separate processor. Again no synchronization is required except for insertions and deletions from $GlobalQ$. In summary, computation is kept local to the extent possible, and synchronization is done only if a load imbalance is detected.

5 Experimental Results

Experiments were performed on a 12 processor SGI Origin 2000 machine at RPI, with 195 MHz R10000 MIPS processors, 4MB of secondary cache per processor, 2GB of main memory, and running IRIX 6.5. The databases we stored on an attached 7GB disk, and all I/O is serial. Further, there were only 8 free processors available to us.

Synthetic Datasets We used the publicly available dataset generation code from the IBM Quest data mining project [IBM,]. These datasets mimic real-world transactions, where people buy a sequence of sets of items. Some customers may buy only some items from the sequences, or they may buy items from multiple sequences. The customer sequence size and transaction size are clustered around a mean and a few of them may have many elements. The datasets are generated using the following process. First N_I maximal itemsets of average size I are generated by choosing from N items. Then N_S maximal sequences of average size S are created by assigning itemsets from N_I to each sequence. Next a customer of average C transactions is created, and sequences in N_S are assigned to different customer elements, respecting the average transaction size of T. The generation stops when D customers have been generated. We set $N_S = 5000$, $N_I = 25000$ and $N = 10000$. Table 1 shows the datasets with their parameter settings.

Parallel Performance In [Zaki, 1998] SPADE was shown to outperform GSP, thus we chose not to parallelize GSP for comparison against pSPADE. As we mentioned in Section 4.0.1, the level-wise idlist data parallel algorithm performs very poorly, resulting in a speed-down with more processors. Results on 1, 2, and 4 processors shown in Figure 7 confirm this.

We now look at the parallel performance of pSPADE. Figure 8 shows the total execution time and the speedup charts for each database on the minimum support values shown in Table 1. We obtain near perfect speedup for 2 processors, ranging as high as 1.92. On 4 processors, we obtained a maximum of 3.5, and on 8 processors the maximum was 5.4. As these charts indicate, pSPADE achieves good speedup performance. Finally, we study the effect of dynamic load balancing on the parallel performance. Figure 7 shows the performance of pSPADE using 8 processors on the different databases under static load balancing (SLB), inter-class dynamic load balancing (CDLB), and the recursive dynamic load balancing (RDLB). We find that CDLB delivers more than 22% improvement over SLB in most cases, and ranges from 7.5% to 38% improvement. RDLB delivers an additional 10% improvement over CDLB in most cases, ranging from 2% to 12%. The overall improvement of using RDLB over SLB ranges from 16% to as high as 44%. Thus our load balancing scheme is extremely effective.

Scaleup Results Figure 9 shows how pSPADE scales up as the number of customers is increased ten-fold, from 1 million to 10 million (the number of transactions is increased from 5 million to 50 million, respectively).

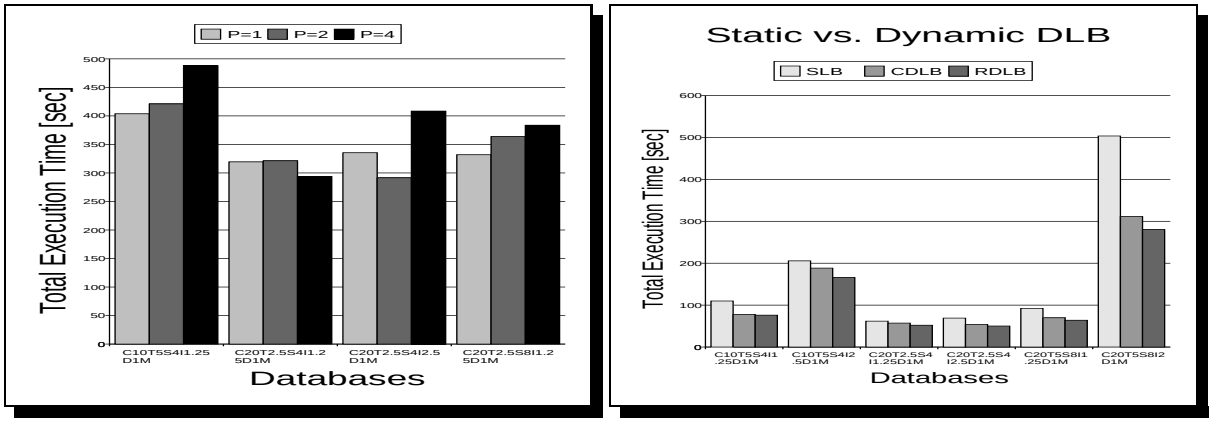


Figure 7: A) Level-Wise Idlist Data Parallelism, B) Effect of Load Balancing

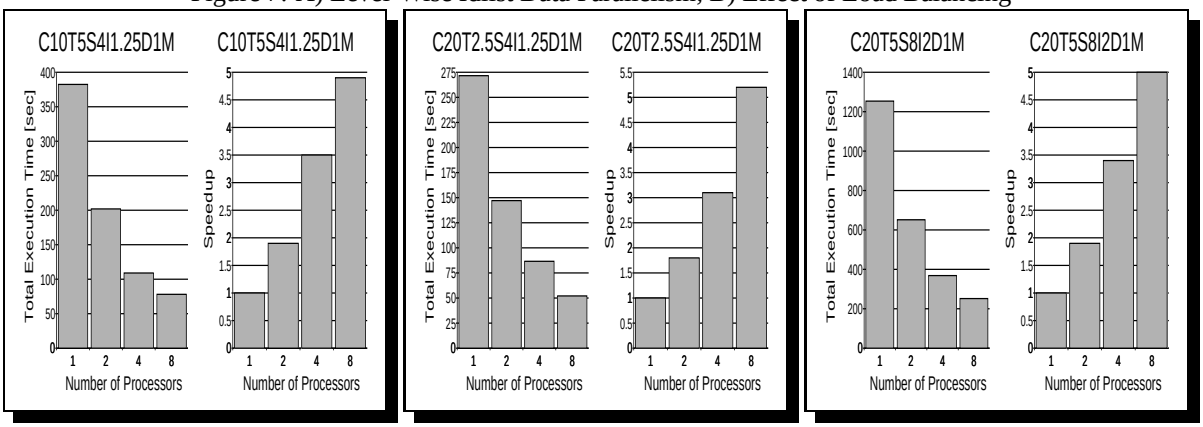


Figure 8: pSPADE Parallel Performance

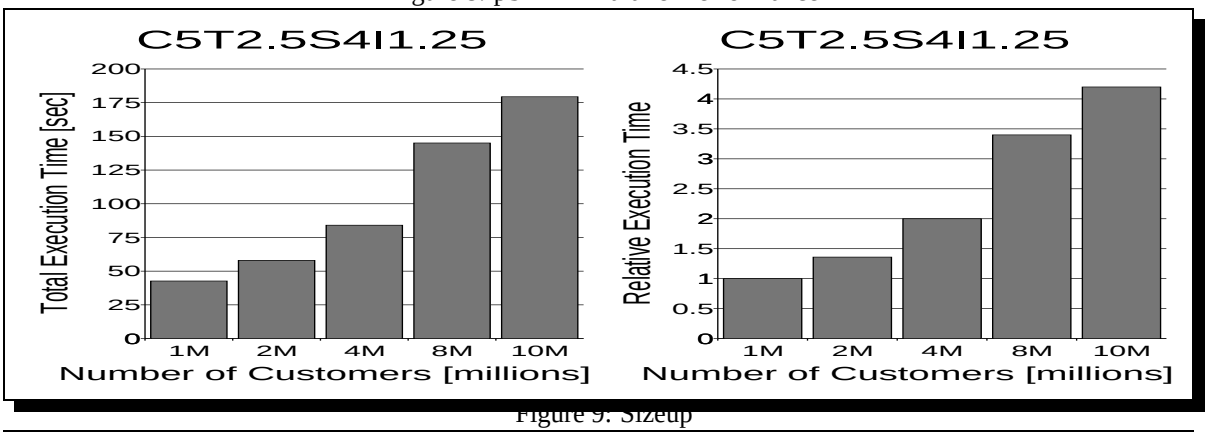


Figure 9: Sizeup

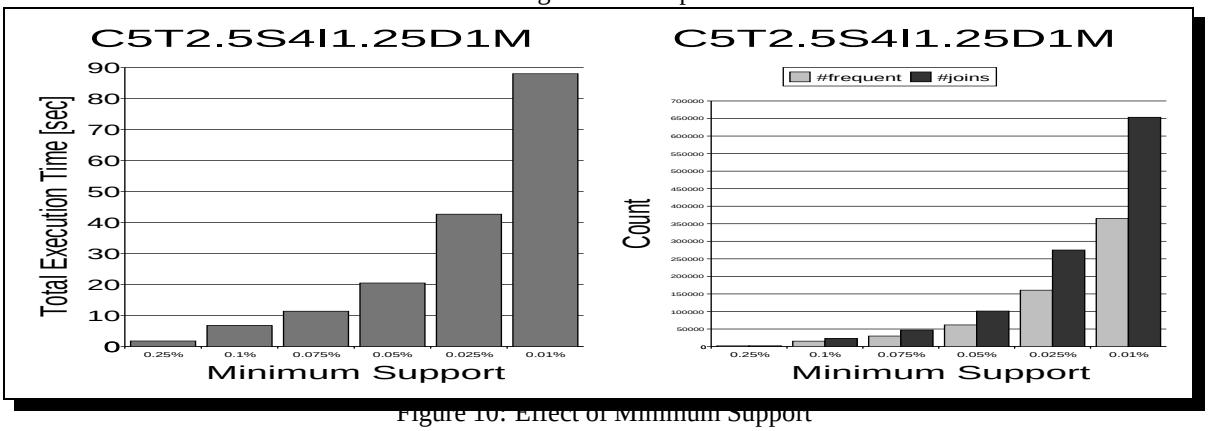


Figure 10: Effect of Minimum Support

The database size goes from 110MB to 1.1GB. All the experiments were performed on the C5T2.5S4I1.25 dataset with a minimum support of 0.025%. Both the total execution time and the normalized time (with respect to 1M) are shown. It can be seen that while the number of customers increases ten-fold, the execution time goes up by a factor of less than 4.5, displaying sub-linear scaleup.

Finally, we study the effect of changing minimum support on the parallel performance, shown in Figure 10. We used 8 processors on C5T2.5S4I1.25D1M dataset. The minimum support was varied from a high of 0.25% to a low of 0.01%. Figure 10 also shows the number of frequent sequences discovered and the number of joins performed (candidate sequences) at the different minimum support levels. Running time goes from 6.8s at 0.1% support to 88s at 0.01% support, a time ratio of 1:13 vs. a support ratio of 1:10. At the same time the number of frequent sequences goes from 15454 to 365132 (1:24), and the number of joins from 22973 to 653596 (1:29). The number of frequent/candidate sequences are not linear with respect to the minimum support.

6 Conclusions

In this paper we presented pSPADE, a new parallel algorithm for fast mining of sequential patterns in large databases. We carefully considered the various parallel design alternatives before choosing the best strategy for pSPADE. These included data parallel approaches like idlist parallelism (single vs. level-wise) and join parallelism. In the task parallel approach we considered different load balancing schemes such as static, dynamic and recursive dynamic. We adopted the recursive dynamic load balancing scheme for pSPADE, which was designed to maximize data locality and minimize synchronization, by allowing each processor to work on disjoint classes. It also has no false sharing. Finally, the scheme minimizes load imbalance by exploiting both inter-class and intra-class parallelism. Experiments conducted on the SGI Origin CC-NUMA shared memory system show that pSPADE has good speedup and scaleup properties.

References

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

[Holsheimer *et al.*, 1996] M. Holsheimer, M. L. Kersten, and A. Siebes. Data surveyor: Searching the nuggets in parallel. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*. AAAI Press, Menlo Park, CA, 1996.

[IBM,] IBM. *http://www.almaden.ibm.com/cs/quest/-syndata.html*. Quest Data Mining Project, IBM Almaden Research Center, San Jose, CA 95120.

[Mannila *et al.*, 1997] H. Mannila, H. Toivonen, and I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery: An International Journal*, 1(3):259–289, 1997.

[Oates *et al.*, 1997] T. Oates, M. D. Schmill, D. Jensen, and P. R. Cohen. A family of algorithms for finding temporal structure in data. In *6th Intl. Workshop on AI and Statistics*, March 1997.

[Shintani and Kitsuregawa, 1998] T. Shintani and M. Kitsuregawa. Mining algorithms for sequential patterns in parallel: Hash based approach. In *Pacific-Asia Conf. on Knowledge Discovery and Data Mining*, April 1998.

[Srikant and Agrawal, 1996] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *5th Intl. Conf. Extending Database Technology*, March 1996.

[Zaki *et al.*, 1997] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. Parallel algorithms for fast discovery of association rules. *Data Mining and Knowledge Discovery: An International Journal*, 1(4):343-373, December 1997.

[Zaki, 1998] M. J. Zaki. Efficient enumeration of frequent sequences. In *7th Intl. Conf. on Information and Knowledge Management*, November 1998.

Communicating Data Mining:
Issues and Challenges in Wide Area Distributed Data Mining

Robert Grossman
National Center for Data Mining, University of Illinois at Chicago
Magnify, Inc.
USA

Yike Guo
Imperial College Parallel Computing Centre
Department of Computing, Imperial College
UK

Today's data mining tools are usually stand along applications that can be used effectively by experts to mine small to moderate amounts of data that is centrally warehoused. The challenges to the data mining community are: 1) to simplify data mining so that it is accessible to non−experts, 2) to scale data mining to large data sets, 3) to scale data mining to data that is widely distributed, and 4) to incorporate data mining into applications, especially decision support applications.

There are three trends that make solving these challenges tractable. First, hardware costs have dropped: workstations and workstation clusters now offer powerful cpus and i/o systems suitable for mining Gigabyte size datasets. Second, high performance networks over a hundred times faster than traditional ethernet are now common in labs and across campuses and with the Next Generation Internet (NGI) are beginning to connect distributed sites. Third, the web is becoming powerful enough that not only can it provide an appropriate infrastructure for viewing distributed multimedia documents but also for analyzing distributed data.

Large−scale data sets are usually logically and physically distributed, and organizations that are geographically distributed need a de−centralized approach to decision support. With the newly available NGI infrastructure and technology, it is feasible for the first time to build up a global knowledge network as an open environment for distributed data mining services. Such an environment will have significant applications in many areas such as scientific knowledge discovery and e−commerce. Moreover, it can be used to mine data generated by embedded, mobile and ubiquitous computing devices. We call such an open knowledge network for managing, mining and modeling large, massive and distributed data sets across WANs a Wide Area Distributed Data Mining System (WADDMS).

In this talk, we will present our joint research on developing technologies and infrastructures for building wide area distributed data mining systems. We will address important issues such as strategies for data and model partitioning, model integration and

multi−table mining. Also we will survey the new network computing technology enabling the wide area data mining. We will also present research on designing protocols for wide area distributed data mining.

As a concrete example, the Terabyte Challenge 2000 Testbed (www.ncdm.uic.edu) will be presented. This is an open, distributed testbed for experimental studies and demonstrations involving data mining, predictive modeling and data intensive computing. The testbed consists of high performance computers and high performance workstation clusters in a dozen cities, including four international sites. Several of the sites are connected by an OC−3 network. The testbed employs a common software infrastructure and is one of the first examples of a global knowledge network.

Efficient Parallel Classification Using Dimensional Aggregates

Sanjay Goil Alok Choudhary
Department of Electrical & Computer Engineering
Northwestern University,
Technological Institute,
2145 Sheridan Road, Evanston, IL-60208
{sgoil,choudhar}@ece.nwu.edu

Abstract

Multidimensional aggregates are frequently computed to improve query performance in Online Analytical Processing applications. We present a new method for decision tree based classification trees using the aggregates computed in the multidimensional data model. The structure imposed on data in a explicit multidimensional storage mechanism leads to efficient dimensional operations. Decision tree based classification algorithms perform computations to find the best split point at each node of the tree. Efficient computation of the split in the decision tree can be done by using the one-dimensional aggregates if the cell values are the class-id values, and counts are maintained for each class. This is used repeatedly at the nodes of the decision tree to calculate splits and manage data. Previous parallel approaches for decision-tree based classification use sorted attribute lists and hash tables to compute the split point and split the data appropriately. The amount of data communicated is proportional to the product of number of records in the training set, and the number of dimensions, at each level of the tree, in the worst case. Parallel formulation of our approach uses data communication proportional to the product of the sum of cardinality of all dimensions and the number of non-classified nodes at each level of the tree. Communication volume is greatly reduced in our approach and is done in one phase of communication at each level of the tree, by coalescing messages. Preliminary results from our experiments on a coarse-grained, distributed memory parallel machine (IBM-SP2) show good performance.

1 Introduction

Classification of large data sets has received considerable attention in the data mining literature recently. The objective of classification is to build a model of the classifying attribute based upon the other attributes of the record. A set of sample records called the training data set is given, consisting of several attributes. Attributes can either be *continuous*, if they come from an ordered domain, or *categorical*, if they are from an unordered domain. One of the attributes is the *classifying* attribute that indicates the *class* to which the record belongs. Several classification models have been used in the past, notably neural networks [Michie *et al.*, 1994], genetic algorithms [Goldberg, 1989], and decision trees [Breiman *et al.*, 1984, Quinlan, 1993]. Among these models, the decision tree models are considered to be the most useful in the domain of data mining because they are relatively inexpensive to construct, easy to interpret and easy to integrate with data base systems. Also, for a variety of problem domains they yield comparable or better accuracy as compared to the other models [Michie *et al.*, 1994].

A decision tree recursively partitions the training set until each partition consists entirely or dominantly records from one class. Each *non-leaf* node of the tree contains a split point which is a test on an attribute and determines how the data is partitioned. Once the decision tree is built from the training set it can be used to classify future instances. The decision-tree based classifiers that can handle large data sets are important because use of larger data sets improves the classification accuracy [Michie *et al.*, 1994].

Previous work in classifying large data sets has been to use sampled data sets or multiple partitions of the data set [Michie *et al.*, 1994, Chan and Stolfo, 1993]. Recent work has focused on using the entire data set, in classifiers like SLIQ [Mehta *et al.*, 1996] and SPRINT [Shafer *et al.*, 1996]. A parallel classifier in the same spirit has been developed in ScalParC [Joshi *et al.*, 1998]. Classifiers like CART [Breiman *et al.*, 1984] and C4.5 [Quinlan, 1993] perform sorting at every node of the decision tree, which makes them expensive for large data sets since disk-based sorting is required at

each node. The approach of SPRINT, SLIQ is to sort the continuous attribute once in the beginning and maintain the sorted order in the subsequent splitting steps. Separate lists are kept for each attribute which maintains a record identifier for each sorted value. In the splitting phase the same records need to be assigned to a node, which may be in a different order in the different attribute lists. A hash table is used to provide a mapping between record identifiers and the node to which it belongs after the split. This mapping is then probed to split the attribute lists in a consistent manner. A framework for instantiating several of these algorithms is presented in [Gehrke *et al.*, 1998], which uses attribute value and class-label pairs (AVC-sets) to make the splitting criteria decision. It is also a greedy top-down approach as the others, except that it works on the AVC-sets at each node of the decision tree. This allows it to use memory more efficiently and perform much better than the attribute-list approaches.

Table 1: Training set data

Row-id	Age	Car-Color	Gender	Class-id
0	10	Green	F	0
1	50	Blue	M	1
2	40	Yellow	F	0
3	30	Green	F	0
4	20	Red	M	1
5	40	Blue	M	0
6	20	Yellow	M	1

Table 1 is an example training set with three attributes, Age, Car color and Gender, and a class attribute. Age is a continuous attribute, whereas both Car color and Gender are categorical attribute. For a categorical attribute having c distinct classes it is assumed that the splitting decision forms c partitions, one for each of its values. Figure 1(a) shows the classification tree for it. At each node the attribute to split is chosen that best divides the training set. Several splitting criteria have been used in the past to evaluate the goodness of a split. Calculating the *gini* index is commonly used [Breiman *et al.*, 1984]. This involves computing the frequency of records of each class in each of the partitions. If a parent node having n records and c possible classes is split into p partitions, the *gini* index of the i^{th} partition is $gini_i = 1 - \sum_{j=1}^{c}(n_{ij}/n_i)^2$, where n_i is the total number of records in partition i, of which n_{ij} records belong to class j. The *gini* index of the total split is given by $gini_{split} = \sum_{i=1}^{p}(n_i/n)gini_i$. The attribute with the least value of $gini_{split}$ is chosen to split the records at that node. The matrix n_{ij} is called the *count matrix*. The count matrix needs to be calculated for each evaluated split point for a continuous attribute.

Categorical attributes have only one count matrix associated with them, hence computation of the gini index is straightforward. For the continuous attributes an appropriate splitting value has to be determined by calculating the $gini_{split}$ and choosing the one with the minimum value. If the attribute is sorted then a linear search can be made for the optimal split point by evaluating the gini index at each attribute value. The count matrix is calculated at each possible split point to evaluate the $gini_{split}$ value. The *gini* index calculations and the node splits for the example above are given in Figure 2. At Node 0, the attribute *Gender* yields the optimal $gini_{split}$ value of 0.214. This creates a split with one partition with M values for gender and another with F values. After this split is made, two child nodes are created. The record values need to be partitioned consistently between the two nodes for the *split-attribute* and the *non-split attributes*. Splitting the split-attribute is straightforward by adjusting pointer values. The challenge is to split the non-split attributes efficiently. Existing implementations such as SPRINT and ScalParC maintain a mapping of the row-id and class-id with the values assigned to each node. The values are split physically among nodes, such that the continuous attribute maintain their sorted order in each node to facilitate the sequential scan for the next split determination phase. A hash list maintains the mapping of record ids to nodes. The record ids in the lists for non-splitting attributes are searched to get the node information, and perform the correct split.

2 Proposed Classification using Multidimensional Aggregates

Multidimensional analysis, OLAP queries and association rule mining are performed efficiently using the materialized aggregates in the data cube. A multidimensional chunk based infrastructure for OLAP and multidimensional analysis for high dimensional data is developed in [Goil and Choudhary, 1998], which optimizes the building of the data cube operator [Gray *et al.*, 1996]. In this model an attribute is treated as a dimension, and records are points in a multidimensional space. Dimensional operations can be performed more efficiently in such a model since a structure is imposed in the storage of data. Multidimensional arrays are the most intuitive and simple structures for this. However, data sets with large dimension cardinalities and a high number of dimensions cannot be handled using arrays. Also, most data sets are sparse and multidimensional arrays lead to redundant storage in such a scenario. We have used a chunk based implementation to sparse data in a bit-encoded sparse structure (BESS) which encodes the indices of the element in a chunk. Dimensional operations are efficiently performed on compressed chunks, which allow a large number of dimensions to be used.

We propose that classification trees can be built using structure imposed on data using the multidimensional data model. Gini index calculation relies on the count

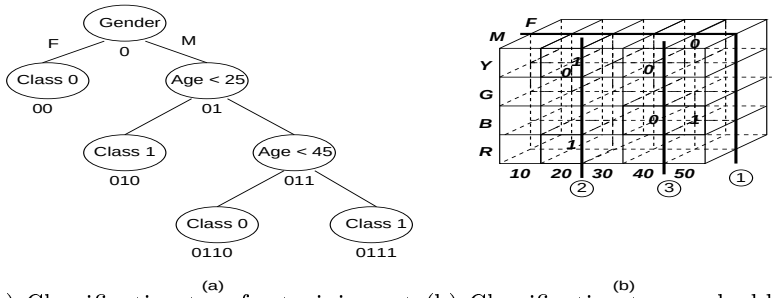


Figure 1: (a) Classification tree for training set (b) Classification tree embedded on a cube

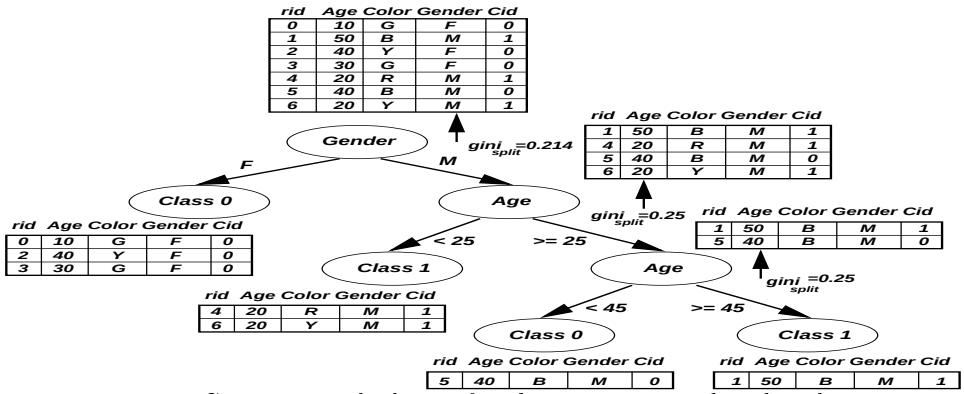


Figure 2: Gini index calculation for the attributes and node splitting

matrix which can be efficiently calculated using the dimensional model. Each populated cell represents a record in the array. For the base cube (which is a multidimensional representation of the records without any aggregation) the class value of the record is stored in each cell. The gini index calculation uses the count matrix which has information about the number of records in each partition belonging to each possible class.

To evaluate split points for a continuous attribute the $gini_{split}$ needs to be evaluated for each possible split point in a continuous attribute and once for a categorical attribute. This means the aggregate calculations present in each of the 1 dimensional aggregates can be used if they have number of records belonging to each class. Therefore for each aggregate we store the number of records in each class. Figure 3(a) gives an example training set with two dimensions, A, a continuous dimension and B a categorical dimension and two class values 0 and 1.

Figure 3(b) is the corresponding multidimensional model. The continuous dimensions A is stored in the sorted order. The aggregates store the number of records mapping to that cell for both classes 0 and 1. To calculate the $gini_{split}$ for the continuous attribute attribute A it is now easy to look at the A aggregate and sum the values belonging to both classes 0 and 1 on both sides of the split point under consideration to get the count matrix. Gini index calculation is done on an attribute list which in the case of a multidimensional model is a dimension. Count matrix is repeatedly calculated on the sorted attribute list which is readily available in the cube structure as a higher level one dimensional aggregate. Each dimension is sorted in the dimensional structure as shown in Figure 3(b).

Figure 4 illustrates the classification tree building process using the multidimensional model and the aggregates maintained at the highest level of the cube structure, one for each dimension.

The challenge is to calculate the one dimensional aggregates efficiently and keep them updated to reflect the partitions after each split. The simple method of computing each level 1 aggregate is to do so from base data. Some alternative strategies that enumerate some intermediate values to optimize the computation of the one-dimensional aggregates have been described in [Goil and Choudhary, 1999].

3 Related Work

In this section we will briefly discuss the previous efforts in parallel classification [Fifield, 1992, Shafer *et al.*, 1996, Joshi *et al.*, 1998] on distributed memory machines, significant of which are parallel SPRINT [Shafer *et al.*, 1996] and ScalParC [Joshi *et al.*, 1998] since they do not require resorting at each node. Classification on fine-grained shared memory machines

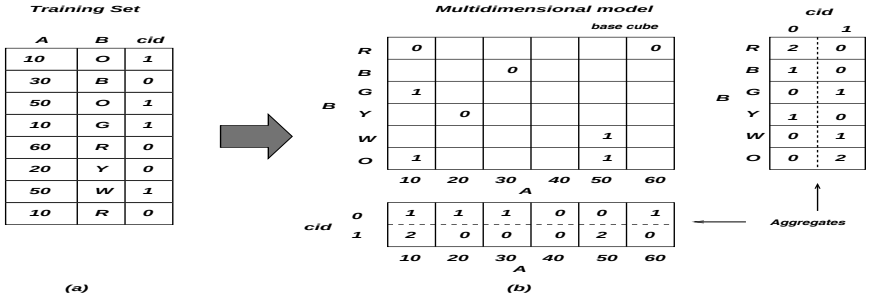


Figure 3: (a) Training set records (b) corresponding multidimensional model

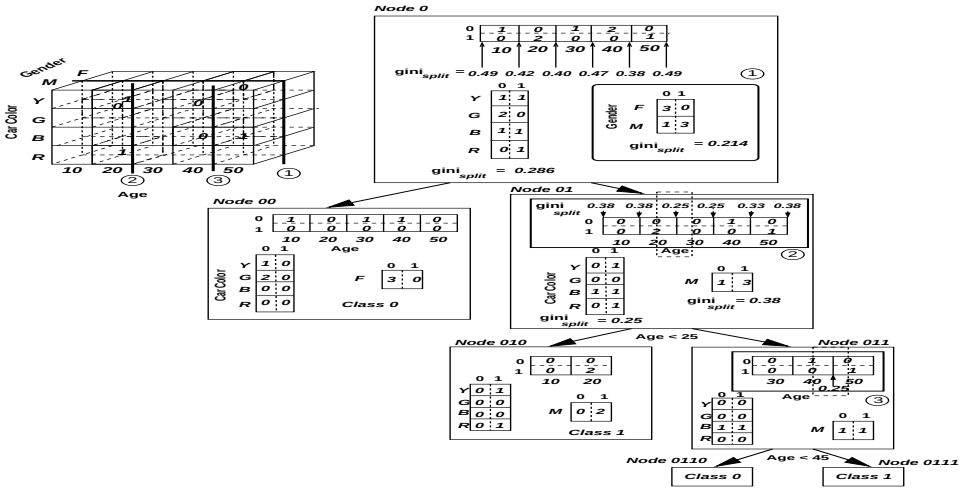


Figure 4: Gini index calculation for the attributes and node splitting with multidimensional aggregates

has addressed in [Zaki *et al.*, 1999]. The issues involved on such architectures are much different than distributed memory machines and hence we do not discuss that work in this paper.

Parallelization of tree methods, especially multidimensional binary search trees (k-d trees), quadtrees and octrees for hierarchical methods have received considerable attention in the areas of scientific processing [Alfuraih *et al.*, 1996]. Classification trees fall under a similar paradigm.

The parallel version of SPRINT partitions each attribute list by sorting each one using a parallel sort with probabilistic splitting. This gives approximately equal sections to each processor of each sorted attribute list. Split points are found by first doing a **prefix sum** operation for the count of values below and above the first split point on each processor followed by building the count matrices locally on each processor. After calculating the gini index locally, the processors communicate to determine which split point has the lowest value. Since there is only one count matrix for categorical attributes, they are constructed locally and one processor collects the global count and calculates the gini values.

Each processor splits its splitting attribute list locally. *rids* are collected from all processors to build the hash table on each processor. The non-splitting attribute lists then probe this table with *rids* to determine the split locally. This has a space complexity of $O(N)$ and makes it unscalable in memory requirements. ScalParC on the other hand maintains a distributed hash table for the splitting phase which has a space complexity of $O(N/P)$. However, the splitting phase is slightly different in their case. A distributed hash table, called the *node table* is maintained by hashing a *rid* with a hash function $h(rid) = (p = rid\ div\ N/P, l = rid\ mod\ N/P)$, where the first field is the processor number and the second is the local index on that processor. After the splitting decision, each processor uses the *rids* of the split attribute to construct hash buffers with $(l, child)$ entry destined for all processors p calculated by the hash function. An **all-to-all personalized** communication phase exchanges these to update the distributed node table. For each non-split attribute list, the distributed node table is queried by filling out a communication buffer with *rids* (enquiry buffer) and sending it to the processor p which then fills it with the child label and sends it back. Two **all-**

to-all personalized communication phases are needed to achieve this.

The size of the node table at each level is usually of size $O(N/P)$. Also, each processor sends $O(N/P)$ elements from each of the non-splitting attribute list. However, there are cases when a processor has to hash all global *rids* at some level and the other processors need to send $O(N)$ elements to be queried by the node table on a single processor [Joshi *et al.*, 1998]. Hence, the worst case complexity is $O(N)$.

4 Parallel Classification using Multidimensional Aggregates

Parallel classification on the multidimensional cube is similar to the sequential classification algorithm, except the fact that each processor calculates the aggregates locally and then needs to update counts for each partitioned dimension from other processors. This is done by a communication phase which calculates a **prefix sum** on the first split point for continuous attributes since each split point is evaluated by computing the $gini_{split}$. For categorical attributes, however, a processor can sum the counts for a value of the categorical dimension and calculate the gini index. Note that separate processors can compute the gini value for a categorical attribute.

4.1 Dimension partitioning and gini index calculation

Figure 5 shows a one dimensional partitioning of the multidimensional base cube and the associated local aggregate calculations. Each processor builds the aggregate locally and for each continuous attribute determines the gini index for the values that lie in its partition. For non-distributed dimensions, each processor locally calculates the aggregates and then does a **reduce all (sum)** on the aggregates for each such dimension. Each processor then works on a part of the aggregate list to calculate the gini index values. This results in computation being partitioned between processors. If the size of the dimension i is d_i, then each processor $P_j, 0 \leq j < P$, calculates the gini index from $(d_i/P)P_j$ to $(d_i/P)P_{j+1}$ for each dimension. For categorical attributes also the computation can be distributed similarly if a **reduce all** is performed on the aggregated dimensions. Otherwise a **reduce (sum)** operation gets the aggregates on a single processor which does the calculation. Since dimensions with categorical attributes are usually small we use the latter approach.

Another alternative is to perform a two dimensional partitioning by selecting the two largest dimensions and partitioning the multidimensional base cube on a 2-dimensional processor grid as shown in Figure 5. This may provide better load balancing in some cases. The communication pattern for the two distributed dimension changes but remains the same for the rest. The non-distributed dimensions get the aggregates locally and do a **reduce all (sum)** to sum each dimensional aggregate independently on each processor. Each processor then works at calculating the gini index in contiguous portions as in the one-dimensional partitioning case. For the distributed dimension A in the figure the reduce for the dimension on processor P_j is done on processor $(P_j/P_Y)P_Y$, where P the number of processors is divided into a two dimensional grid in the dimensions X and Y as $P_X \times P_Y$. The processors $P_0 + i * P_Y, i = 0, \ldots P_X$ then calculate the local sums of class ids and need a **prefix sum** to get the values updated across processor boundaries with the number above and below in each class. Each of these processors then calculates the gini index for the local dimension values.

Similarly, for dimension B a **reduce** operation is done on processor $P_j \% P_Y$ and again a **prefix sum** is done between processors $P_0 + i, i = 0, \ldots P_Y$ to calculate the gini index locally.

4.2 Node splits

Once the $gini_{split}$ is calculated for all attributes (dimensions), each processor picks out the minimum and the attribute it is related to. A **Reduce (minimum)** gets the minimum and the attribute which is the *split* attribute. Each processor then partitions the split attribute locally. Each processor maintains a mapping of dimension indices to the node they belong to, as they are split. In other words, a global tree representation is present on each node. Each node keeps the dimensional boundaries of the dimensions split, defining a hypercube the node encapsulates. The aggregates are then calculated within the node boundaries. The communication required is for each dimension, but now there are multiple count values, one for each active node which has not yet been classified.

Let n_a be the number of attributes (dimensions) divided into n_d, the set of distributed dimensions and n_u, the set of non-distributed dimensions. For non-distributed dimensions these need to be aggregated using a **Reduce (sum)** on all processors to distribute the computation of the gini index calculation. The amount of data communicated at a level k of the classification tree is $O(\mathcal{N}_n^k |D_u|)$, where $\mathcal{N}_n^k$ is the number of active nodes at a level k, and $|D_u|$ is the sum of dimension sizes of dimensions $d_i \in n_u$. For a distributed dimension a local aggregation is done for each dimension for each node and then a **Parallel prefix (sum)** is done. The data communicated for this operation at level k is $O(\mathcal{N}_n^k n_d)$. The total data communicated at level k of the classification tree is then $O(\mathcal{N}_n^k(|D_u| + n_d)) = O(\mathcal{N}_n^k |D_a|)$, $|D_a| = |D_u| + |D_d|$, because the number of split dimensions is usually one or

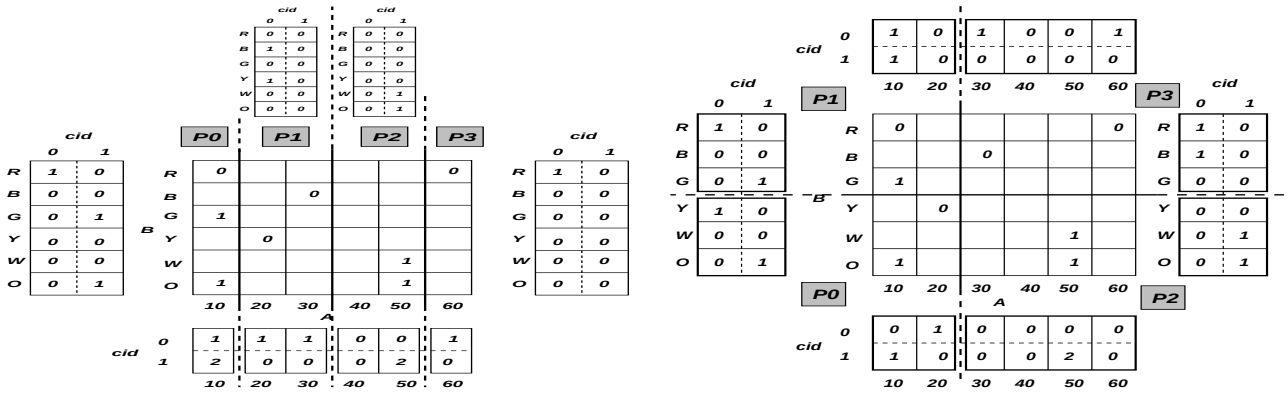


Figure 5: One and two dimensional partitioning of dimensions

two. This is the case for our implementation where the optimization of message-coalescing, reduces the number of communication messages by combining the messages as one buffer.

For the scalable record based classification algorithm (ScalParC) the complexity of communication at each level is $O(n_a \frac{N}{P})$ at each level and $O(n_a N)$ in the worst case, where N is the size of the training set. The number of active nodes at any level is much smaller than the size of the training set. Also, the sum of dimension sizes, $|D_a|$, is also much smaller when compared to N. Thus, $\mathcal{N}_n^k \times |D_a| < N$, for a large N. This makes the overall communication requirement of our multidimensional classification algorithm better. Computation of the 1-D aggregates and the gini index calculations has the time complexity $O(\frac{N}{P} + \mathcal{N}_n D_a)$ in our method, whereas ScalParc requires $O(\frac{N}{P} n_a)$, to compute the gini indices and $O(N n_a)$ in the worst case.

A **broadcast** of the prefix sum from the last processor is followed by the local calculation of the gini index and a **Reduce (minimum)** for the gini value which determines the attribute used for the split. Notice that for a non-distributed categorical attribute B the calculation is distributed across processors by letting each processor work on a section since all information is available on each processor as a result of the reduce. Suppose the $gini_{split}$ results in selecting $A = 25$ as the split point in the example.

Figure 6 shows the split at each processor into L and R nodes labeled 00 and 01 respectively. The counts of each dimension are done for each node on each processor. The steps for parallel gini index calculations are done for each node on every processor. Communication can be concatenated for the nodes at each level following the idea of concatenated parallelism in [Al-furaih *et al.*, 1996] for each active node.

4.3 Communication Optimizations

Only one collective communication operation is performed for all nodes at each level. This is due to the fact that information that needs to be communicated for all nodes is stored contiguously. Each node has the counts of class id. values for each dimension. This is combined using a **Reduce (sum)** operation across all nodes as described in a previous section. The count matrices for each dimension are allocated contiguously for each active node as shown in Figure 7. Each node allocates memory for each dimension, d_i, for each class. At each level only the unclassified nodes (active nodes) are represented and need to participate. Algorithm 1 describes the steps of the overall parallel algorithm for classification.

5 Performance Results

We use the synthetic data generator introduced by Agrawal et al. in [Agrawal *et al.*, 1993]. It is a widely used synthetic set used by many others, primarily because there are no large real data sets available.

The synthetic set has nine attributes as shown in Table 2. $\mathcal{U}(x : y)$ denotes the integer uniform distribution with values $v : x \leq v \leq y$. The data set can be generated with different classification functions that assign labels to the records produced. We use Function 1, which is the default, for our performance study. We have used various subsets of the data to evaluate the performance of our algorithm, varying the number of dimensions, the number of records and the classification threshold. We observe good speedup and scale-up performance for our experiments performed mainly on a 16 node IBM SP2 with thin nodes (120Mhz processor) running AIX and having 128 MB main memory per processor. We present a subset of our preliminary results in this section to illustrate performance. More results will be given in the final version of the paper.

Figure 8(a),(b) shows the time for classifying 1 million and 5 million records with 8 dimension (again

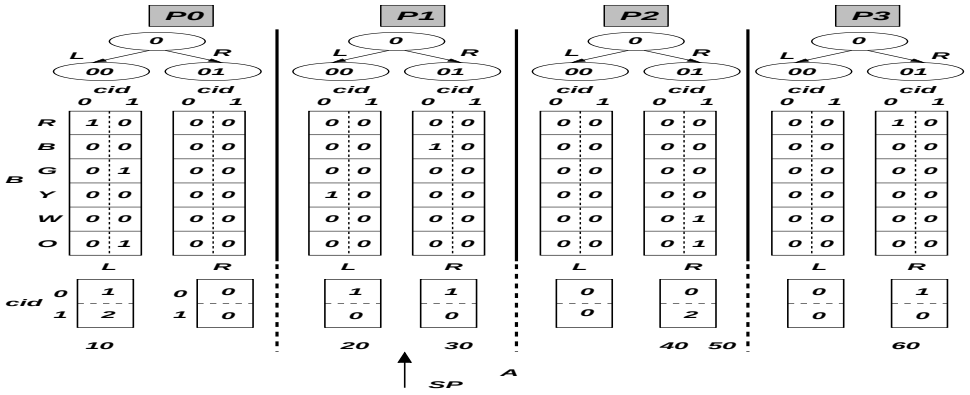


Figure 6: Node split of dimension $A = 25$, and the count arrays for each node

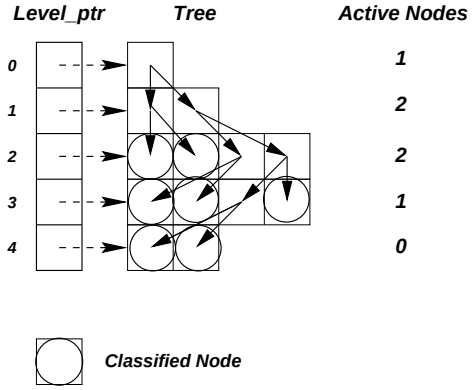


Figure 7: Tree structure at each processor, nodes being stored contiguously to coalesce messages for communication at each level

Algorithm 1 *Multidimensional Classification Algorithm*

/* set the root node to represent the entire multidimensional space */
for $i \leftarrow 0$ to k
    $begin_i = 0$
    $end_i = |d_i|$
$n\_active\_nodes = 1$;
while($n\_active\_nodes > 0$)
    For each chunk c
        Get chunk boundary values in each dimension
        For each active node $j, (0 \leq j < n\_active\_nodes)$
            Determine if chunk c is contained within node j,$[begin_i, end_i)$, for all i, else if it overlaps or else not in range.
    For each active node $j, (0 \leq j < n\_active\_nodes)$
        Compute the contribution of chunk c to 1-dimensional aggregate if the chunk is inclusive or overlapping to the node j.
    One **Reduce (sum)** communication operation for consolidating k 1-dimensional aggregates on all processors.
    For each active node $j, (0 \leq j < n\_active\_nodes)$
        Check if node j is already classified within the threshold level.
        If the node is *not* classified
            For each attribute $i, 0 \leq i < k$,
                Compute *gini* indices for each dimension i recording the minimum value and corresponding i ($split_i$) and the split point.
            Update for the child node, $begin_i$ and end_i for $split_i$ using the split point and copy the others from the parent node j.
            Add the number of child nodes to the count of active nodes.
    Update n_active_nodes as the count of child nodes.
end

Table 2: The sizes of the dimensions in data set used

Predictor Attribute	Distribution	Max. number of entries
Salary	$\mathcal{U}(20K, 150K)$	131
Commission	0, if(salary $>$ 75K) else $\mathcal{U}(10K, 75K)$	66
Age	$\mathcal{U}(20, 80)$	61
Education	$\mathcal{U}(0, 4)$	5
Car	$\mathcal{U}(1, 20)$	20
Zip Code	$\mathcal{U}$(nine zip codes)	9
House Value	$\mathcal{U}(0.5 \times k \times 100K, 1.5 \times k \times 100K)$ k depends on ZipCode	1351
Home Years	$\mathcal{U}(0, 30)$	31
Loan	$\mathcal{U}(0, 500K)$	501

without Loan attribute) and classification threshold value set to T = 0.8. The 5 million records data set is We observe that most of the time is taken for the 1 dimensional aggregate calculations and the gini computations, a small fraction taken by communication and maintenance of the classification tree.

Figure 8(c) shows the classification performance for a 9 dimensional data set with 1 million records. The major component of the computation is the 1-dimensional aggregate calculations for all nodes. This is the parallel component and we see good speedup as the number of processors is increased. The other phases of communication and gini index calculations as shown in the figure are small portions of the entire time. Communication increases slightly as number of processors increase, due to the increase in p term in the complexity of the **Reduce** global communication operation. We are currently running experiments with larger data sets and will include the results in the final version.

6 Conclusions

In this paper classification using decision trees is performed on an explicit multi-dimensional storage scheme. The split point for a node in the classification tree is chosen after computing the counts of records belonging to each class for each attribute value, in each dimension. A multidimensional representation make a 1-dimensional computation for each dimension from the base cube very efficient. A parallel framework is used to parallelize this calculation, and one round of communication is required at each level of the tree by concatenating (coalescing) communication for each node together. Results on a synthetic benchmark [Agrawal *et al.*, 1993], widely used for classification algorithms performance, shows good parallel performance.

7 Acnowledgements

This work was supported in part by NSF Young Investigator Award CCR-9357840 and NSF CCR-9509143.

References

[Agrawal *et al.*, 1993] R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, December 1993.

[Al-furaih *et al.*, 1996] I. Al-furaih, S. Aluru, S. Goil, and S. Ranka. Parallel construction of multidimensional binary search trees. In *Proc. International Conference on Supercomputing*, May 1996.

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

[Chan and Stolfo, 1993] P.K Chan and S.J Stolfo. Meta-learning for multistrategy and parallel learning. In *Proc. International Workshop on Multistrategy Learning*, 1993.

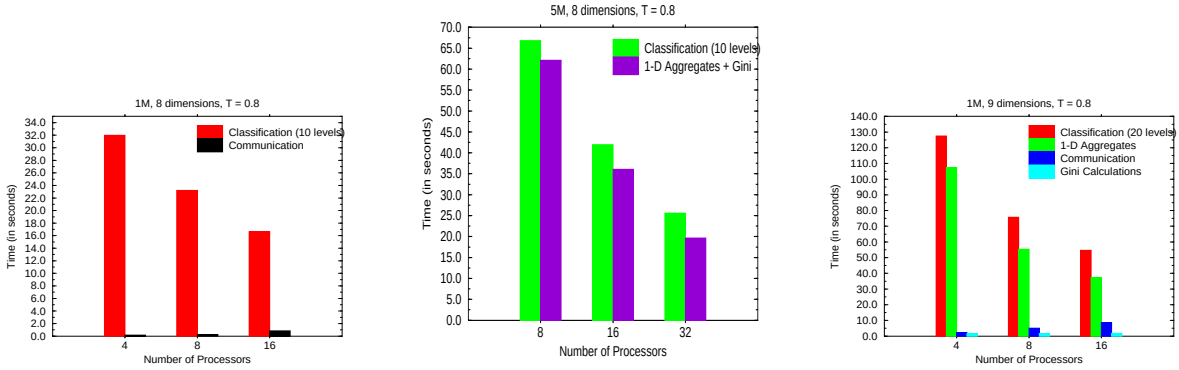
[Fifield, 1992] D.J Fifield. *Distributed Tree construction from large data sets*. Bachelor's Honors Thesis, Australian National University, 1992.

[Gehrke *et al.*, 1998] J. Gehrke, R. Ramakrishnan, and V. Ganti. RainForest - A Framework for Fast Decision Tree Construction of Large Data Sets. In *Proc. 24th International Conference on Very Large Databases*, 1998.

[Goil and Choudhary, 1998] S. Goil and A. Choudhary. High performance multidimensional analysis and data mining. In *Proc. SC98: High Performance Networking and Computing Conference*, November 1998.

[Goil and Choudhary, 1999] S. Goil and A. Choudhary. Parallel classification using the multidimensional data model (under preparation). Technical Report CPDC-9904-006, Northwestern University, January 1999.

[Goldberg, 1989] D.E Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Morgan Kaufmann, 1989.



(a) (b) (c)

Figure 8: Classification of a 8 dimensional data set for 10 levels, with classification threshold T = 0.8 for (a) 1M records (b) 5M records (c) Classification of a 9 dimensional data set for 20 levels, with classification threshold T = 0.8 for 1M records

[Gray *et al.*, 1996] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proc. 12th International Conference on Data Engineering*, 1996.

[Joshi *et al.*, 1998] M. Joshi, G. Karypis, and V. Kumar. ScalParC: A new scalable and efficient parallel classification algorithm for mining large datasets. In *Proc. International Parallel Processing Symposium*, March 1998.

[Mehta *et al.*, 1996] M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A fast scalable classifier for data mining. In *Proc. of the Fifth International Conference on Extending Database Technology*, March 1996.

[Michie *et al.*, 1994] D. Michie, D.J. Spiegelhater, and C.C. Taylor. *Machine Learning, Neural and Statistical Classification.* Ellis Horwood, 1994.

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

[Shafer *et al.*, 1996] J.C. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proc. 22th International Conference on Very Large Databases*, September 1996.

[Zaki *et al.*, 1999] M. Zaki, C.T. Ho, and R. Agrawal. Scalable parallel classification for data mining on shared-memory multiprocessors. In *Proc. International Conference on Data Engineering*, March 1999.

Learning Rules from Distributed Data

Lawrence O. Hall, Nitesh Chawla, Kevin W. Bowyer
and W. Philip Kegelmeyer*
Department of Computer Science and Engineering,
ENB 118
University of South Florida
4202 E. Fowler Ave.
Tampa, Fl 33620
hall@csee.usf.edu

*Sandia National Laboratories
Advanced Concepts Department
P.O. Box 969, MS 9214
Livermore, CA, 94551-0969

Abstract

In this paper a concern about the accuracy (as a function of parallelism) of a certain class of distributed learning algorithms is raised, and one proposed improvement is illustrated. We focus on learning a single model from a set of disjoint data sets, which are distributed across a set of computers. The model is a set of rules. The distributed data sets may be disjoint for any of several reasons. In our approach, the first step is to construct a rule set (model) for each of the original disjoint data sets. Then rule sets are merged until an eventual final rule set is obtained which models the aggregate data. We show that this approach compares to directly creating a rule set from the aggregate data and promises faster learning. Accuracy can drop off as the degree of parallelism increases. However, an approach has been developed to reduce this problem.

1 Introduction

Training data may be distributed across a set of computers for several reasons. For example, several data sets concerning telephone fraud might be owned by separate organizations who have competitive reasons for keeping the data private. However, the organizations would be interested in models of the aggregate data.

Another example is very large datasets that will not fit in a single memory which are useful in the process of learning a classifier or model of the data. It is now possible to have training data on the order of a terabyte which will not fit in a single computer's memory. A parallel approach to learning a model from the data will solve the practical problem of how to deal with learning from large data sets.

This paper describes an approach that learns a single model of a distributed training set in the form of a set of rules. A single model may be an advantage in the case that it will be applied to a large amount of data. For example, consider the problem of visualizing "interesting" regions of a large data set. A set of rules might be learned which can do this. These rules would then be applied to similarly large data sets to guide the user to the interesting regions.

This paper examines an approach to generating rules in parallel that is related to work by [Williams, 1990, Provost and Hennessy, 1996]. A set of rules will be generated from disjoint subsets of the full data set used for training. Given N disjoint subsets of the full dataset there will be N sets of rules generated. Each subset of data may reside on a distinct processor. The distributed rule sets must be merged into a single rule set. Our focus is towards using a large N with very large training sets.

The final set of merged rules should be free of conflicts and have accuracy equivalent to a set of rules developed from the full dataset used for training. We discuss an approach to building a single, accurate set of rules created from N rule sets. The question of how similar to one another rule sets developed sequentially and in parallel might be is explored. Experimental results on several small, representative datasets show that accuracy tends to decline as N increases. A method to reduce this tendency is presented.

In Section 2 the generation of rules in parallel and the combination of rule sets is discussed. Section 3 contains experimental results and a discussion of the issues shown by an analysis of them. Section 4 contains a summary of the strengths and open questions associated with the presented approach to learning in parallel.

2 Generating rules in parallel and combining them

The disjoint subsets of extremely large data sets may also be very large. In principle any approach that produces rules can be used to learn from each data set. It is possible, for example, to learn decision trees [Quinlan, 1992, Quinlan, 1996] in a fast, cost effective manner. Learning a decision tree, pruning it and then generating rules from the pruned tree will be an effective competitor from a time standpoint to other rule generation approaches such as RL [Clearwater *et al.*, 1989] or RIPPER [Cohen, 1995].

In the work reported here, rules are created directly by traversing pruned decision trees (with the obvious optimization of removing redundant tests). The process of creating rules from decision trees in a more time consuming fashion has been covered in [Quinlan, 1992, Kufrin, 1997]. In learning rules it is often the case that a default class is utilized. However, it is desirable to avoid having default classes for examples because the lack of a model for some examples cannot be resolved in a straightforward way when rule sets are merged.

Each rule that is created will have associated with it a measure of its "goodness" which is based on its accuracy and the number and type of examples it covers. We are using a normalized version of Quinlan's certainty

factor [Quinlan, 1987, Provost and Hennessy, 1996] to determine the accuracy of a rule R over an example set E as:

$$acc(R, E) = (TP - 0.5)/(TP + \rho FP), \qquad (1)$$

where TP is the number of true positives examples covered by R when applied to E, FP is the number of false positives caused by R when applied to E, and ρ is the ratio of positive examples to negative examples for the class of the rule contained in the training set.

A rule, R, must have acc(R,E) $\geq t$ for some threshold t in order to be considered acceptable over a set of E examples. When a rule is built on a single subset of data, its accuracy may change as it is applied to each of the other subsets of data. The rule can be discarded whenever its accuracy is less than t or only after it has been applied to all of the distributed examples and has an accuracy below the threshold.

Discarding a rule as soon as it is below the accuracy threshold will save the testing time on other processors and some communication time required to send it and its current TP/FP count to another processor. Testing time is not likely to be very high and communication time for one rule will generally be low. So, the per rule savings may be relatively low. On the other hand a rule which performs poorly on one partition and then improves to be acceptable or quite good will be ruled out under the incremental deletion approach. Our approach will be to only delete rules after testing is complete.

2.1 Merging rule sets generated in parallel

In [Provost and Hennessy, 1994] it is shown that any rule which is acceptable, by the accuracy definition in (1), on the full training set will be acceptable on at least one disjoint subset of the full data. This suggests that a rule set created by merging sets of acceptable rules learned on disjoint subsets of a full training set will contain rules that would be found on the full training set. Earlier work on building rules in parallel and then merging them [Provost and Hennessy, 1996] found that the merged set of rules contained the same rules as found by learning on the full data set and some extras. In this work, the training set was large, over 1,000,000 examples.

However, in Figure 1, we show a small illustrative data set for which the rules learned by merging disjoint rule sets built on a disjoint 2 partition of the data do not include any rules learned by training on the full data set. Information gain is used to choose the attribute test for each node in the decision tree [Quinlan, 1996].

Figure 1 shows that a merged rule set, with each of the constituent rule sets developed in parallel on a disjoint training set, may in the extreme contain no rules in common with the rules created by training on the union of all the subsets (i.e. the full training set). The final merged rule set will depend upon how the examples are partitioned. The mix of examples needs to reflect the mix of available training examples.

The example data set and results from it shown in Figure 1 suggest that the accuracy of merged rules may well be different from the accuracy of the rules created from the full training set. Our experimental results will examine how different the accuracy may be and how it is affected by the number of partitions made from the training data.

As rule sets are merged contradictions may be introduced in the sense that an example may be classified into different classes by two rules. As an individual rule is applied to more labeled examples its accuracy may change significantly. Consider two rules R1 and R2 which classify an overlapping set of examples into two different classes. As the rules are applied to all of the subsets of the original training examples, the accuracy of one of them is expected to become less than a well-chosen threshold, t. Hence, one will be removed and the conflict resolved. However, it is possible for partially conflicting rules to survive.

For example, from the Iris data set [Merz and Murphy,] using 2 partitions we get the rules shown in Figure 2. The final accuracy after they have been applied to all training examples (but learned only from the examples in one partition) is shown in the second set of brackets associated with the rule. Both rules perform quite well when the accuracy measure in (1) is applied to them. If the conflict is not resolved, then rule ordering will affect the rules performance. A reasonable choice might be to give higher priority to the rule with a better value of acc(R,E). Alternatively, conflict can be resolved as shown in [Hall *et al.*, 1998, Williams, 1990]. Essentially conditional tests can be added to one or both rules resulting in specialization of the rules. Any examples left uncovered can then be classified by a newly created rule. Here, we will remove the lowest performing conflicting rule with any uncovered examples being assigned to the majority class.

Another type of conflict occurs when two rules for the same class created from different disjoint subsets have coverage which overlaps. For example, the rules shown in Figure 3, can be combined as the second more general rule. In general when there are overlaps among rules for the same class, the more general test is used.

3 Experiments

The experiments reported here are from two datasets from the UC Irvine database [Merz and Murphy,] both of which consist of all continuous attributes. The IRIS data set [Fisher, 1936] has 150 examples from 3 classes and the PIMA Indian diabetes data set has 768

Examples

name	attr. 1	attr. 2	class
A	1	1	C1
B	4	3	C1
C	3	2	C1
D	2	4	C2
E	5	2	C2
F	6	1	C2

Sorted on Attr. #1

name	attr. 1	class
A	1	C1
D	2	C2
C	3	C1
B	4	C1
E	5	C2
F	6	C2

Sorted on Attr. #2

name	attr. 2	class
A	1	C1
F	1	C2
C	2	C1
E	2	C2
B	3	C1
D	4	C2

Processor 1 gets examples BCD and produces the rules:

```
if attr 1 > 2 then C1.
if attr 1 <= 2 then C2.
```

or

```
if attr 2 > 3 then C2.
if attr 2 <= 3 then C1.
```

Processor 2 gets examples AEF and produces the rules:

```
if attr 1 <= 1 then C1.
if attr 1 > 1 then C2.
```

From the full training set we get the rules:

```
if attr 1 > 4 then C2.
if attr 1 <= 4 and attr 2 <= 3 then C1.
if attr 1 <= 4 and attr 2 > 3 then C2.
```

Figure 1: An example where rules built in parallel on disjoint subsets **must** be different from rules built on the full data set. Using information gain to decide the splits.

```
if petal width in cm > 0.4 and
   petal length in cm > 4.9
then class Iris-viginica [1/23]
                         [1/40]
if  0.5 < petal width in cm <= 1.6
then class Iris-Versicolor [0/23]
                           [4/48]
```

Figure 2: Example of two rules from 1 fold of an Iris data 2 partition which have conflicts but survive to the final set. The numbers in brackets are the false positives and number of examples covered respectively. The second set of numbers for a rule is its accuracy after it is applied to the partition on which it was not learned.

```
a) if x > 7  and x < 15 then Class1
b) if x > 9 and x < 16 then Class1
c) if x > 7 and x < 16 then Class1
```

Figure 3: Two overlapping rules, a and b, can be replaced by the third, c.

examples from 2 classes. We are interested in how the accuracy is affected by partitioning these small data sets into N disjoint subsets, learning decision trees on the subsets, generating rules from the decision trees and then merging the rules into a final set of rules.

Our experiments were done using 10 fold cross validation [Weiss *et al.*, 1990]. For an individual data set and a given number of disjoint subsets, N, 10 partitions of the data were made each consisting of 90% of the train data with a unique 10% held back for testing. From each fold, N disjoint subsets are created. C4.5 is applied to each of the N subsets and rules are created from the generated decision tree. The rules created from the j^{th} subset of data are then applied to the N-1 other subsets. The accuracy of each rule must be greater than the chosen threshold, t, in order for the rule to remain in the final set of rules. The default for the threshold t was chosen as 51, just slightly better than guessing every example belongs to the class of the rule. For the Iris data we chose t=75 rather arbitrarily. Setting t in an appropriate and systematic way must still be addressed.

For the Iris data, we have done an experiment with N=2. With the default C4.5 release 8 parameters the results on Iris for 10-fold cross validation and the results from the approach described here (with 2 different choices for certainty factors or cf's for use in pruning) are given in Table 1. The average number of rules was 6.5 for the default cf=25 and 3.1 for cf=1.

Table 1: Results on the Iris data set using 10-fold cross-validation for a 2 processor partition. sd - standard deviation.

C4.5 % Correct ± sd	Pruned (cf=25) % Correct ± sd	Pruned (cf =1)% Correct ± sd
95.3 ± 6.01	94 ± 6.96	94.7 ± 5.81

The reason for decreasing the certainty factor for pruning was to make the rules produced on the data subsets more general and less likely to overfit on the small number of examples. On this dataset there was a small positive impact.

The results for the simulated parallel approach are insignificantly worse than for C4.5 with default parameters but comparable to C4.5 with the cf=1 (94.7% and std= 5.81%).

A more interesting experiment is to look at a significant number of partitions. With the larger Pima data set experiments were run with N=2, N=4, ..., N=10, and N=12. The results of a 10-fold cross-validation experiment with C4.5 using its default pruning (cf=25) were an average accuracy of 73.90% with sd=4.26% and an average of 23.8 rules. Figure 4 shows plots of accuracy, standard deviation and the number of rules for 10-fold cross validation experiments with each of the above N disjoint partitions. The performance of rules created from the unpruned tree, the pruned tree with the certainty factor of 25 and a certainty factor of 1 are shown. It can be seen that the accuracy of the rule set generally decreases as N increases. The standard deviation tends to get large suggesting that performance on some folds is quite poor. The number of rules that remain in the final set remains fairly constant as N is increased. There are significantly less rules, after conflict resolution, than when training is done on the full data set.

3.1 Discussion

The results obtained here come from small datasets. However, we believe the issue of rule accuracy falling off can also occur with larger datasets. Our results are consistent with those found in [Chan and Stolfo, 1996] where experiments were conducted on data sets related to the human genome project. It was found that more partitions resulted in lower performance on unseen data. Their approach to combining the classifiers was different, relying on voting among the learned set of classifiers.

Some success in mitigating the effect of learning in parallel may be gained by using a combiner or arbitrator approach to integrating multiple classifiers [Stolfo *et al.*, 1997, Chan and Stolfo, 1998]. However, such approaches entail retaining all N classifiers learned in parallel and may be problematic for large N. There is

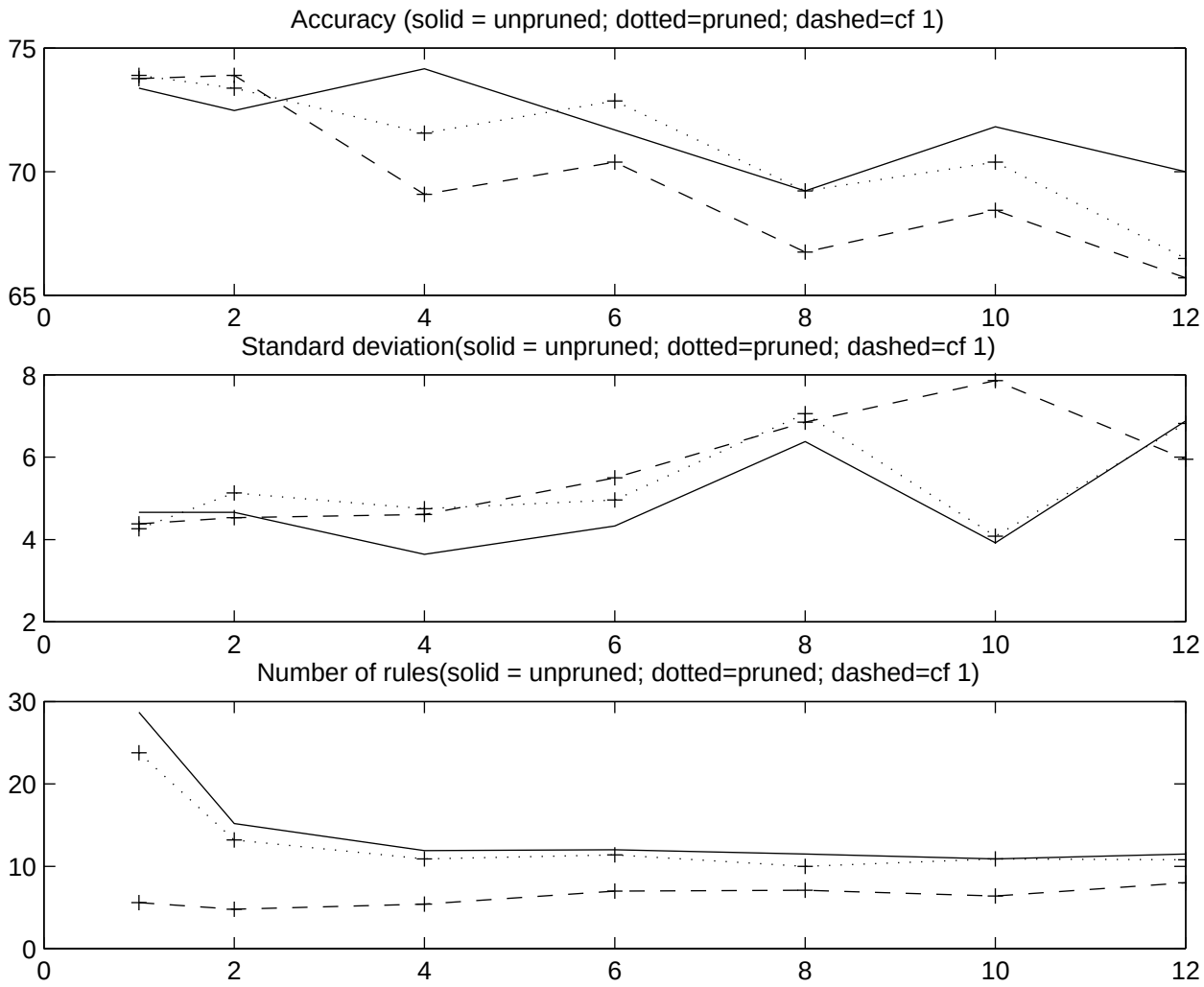


Figure 4: Results from PIMA dataset experiments.

not a single model of the data either.

In [Provost and Hennessy, 1996] an approach similar to ours was used on a very large dataset (over 1,000,000 examples) and there was no drop off in accuracy for partitions up to N=4. Our results suggest that accuracy would fall off as N increased.

If there are enough representative examples of each class in each of N disjoint partitions, the combined ruleset will have high accuracy. Clearly, the limit case is that each of the N subsets has an example which exactly or in the case of continuous data, almost exactly, matches each of the examples in the other subsets. So, the data really consists of only $|S_i|$ distinct examples, where $|S_i|$ is the set of examples at the i^{th} compute node.

The very worst case is that $|S_i| = C$, the number of classes in the data. In this case each subset consists of just one example of each class. Clearly, this is an unreasonable choice for N and no one would make it in practice.

Under the approach to parallel rule generation covered here there is the usual question of how large N can be before communication costs begin to slow the rule generation process significantly. However, there is the more important question of determining an N for which the accuracy of the resultant rule set is acceptable. In datasets that are too large to learn from on a single processor, it will not be possible to know what the maximum accuracy is.

Clearly with this approach a tradeoff between accuracy and speed exists. The use of more processors promises that each can complete its task faster on a smaller training set at the usual cost of coordinating the work of all the processors and waiting for the combination of rules to be completed. However, there is a second *accuracy* cost that will be paid at some point as N becomes large. What the point of significant accuracy falloff is and how to recognize it is an open question.

3.1.1 Improving highly parallel performance

On very small datasets, the rules learned will tend to be too general. A good rule on one dataset may prove to wrongly classify many examples on another processor which belong to a different class. Specializing such rules by adding conditional tests to them can help rule out some or all of the examples that are incorrectly classified on a given processor by a rule created on a different processor.

In a couple of small experiments, we have adopted the following strategy to improve rule performance. Any rule that is within 5% of the acceptable threshold and was created on a processor other than the current processor is a candidate for specialization. The rule is specialized by taking the examples that are classified by the rule and growing a decision tree on them. Then one takes the test(s) along the best branch and adding this to the rule to create a new specialized rule. The specialized rule as well as the original rule will be tested against the next subset of examples. The accuracy of the specialized rule is only reported on the examples available on the current processor. Both the original rule and the specialized rule can be further specialized as new data is encountered as long as their performance remains within 5% of the threshold, t.

A good feature of this approach is that there will be no decrease in performance as long as the original rule remains in the final rule set, but has a lower priority than its specializations. Any examples left uncovered by the specialized rules will still be classified by the more general rule. Of course, the general rule will only exist at the end if its accuracy value is above the threshold.

As a simple example of the potential for improvement consider the example of Figure 1. Assume you got the second set of rules using attribute 2 (Att2) from processor 1 (Proc1) and applied them to the examples held by processor 2. The rule

```
if Att2 <= 3 then C1
```

now gets only 2/5 examples correct. If it is specialized to be

```
if Att2 <= 3 and Att1 < 5  then C1
```

the rule will cover 3/3 examples correctly. Further it essentially matches the second rule obtained from the whole training set in Figure 1. This example shows how specialization would work.

On the Pima data, specialization was applied to the two partition case raising the accuracy slightly from 73.38% to 73.77%.

Rule specialization could be decided upon in other ways than in our experiment. For example, after learning a pessimistic estimate of the rules performance [Quinlan, 1992] could be generated. For a test subset on which the classification performance of the rule was more that x% below the estimate, specialization could be carried out. To get a better estimate of the performance of a specialized rule, it might be tested against all the data on which it was not created (e.g. broadcast to all processors). This would make the conflict resolution process more accurate.

4 Summary

This paper discusses an approach to creating rules in parallel by creating disjoint subsets of a large training set, allowing rules to be created on each subset and then merging the rules. It is shown that this approach can provide good performance. It is also pointed out that the rules discovered in parallel may be different from those discovered sequentially. While it is true that rules

which perform well on the full data set will perform well on at least one subset of the data, it is not necessarily the case that these rules will be discovered.

In an empirical study, it is shown that the accuracy of the merged rule sets can degrade as the number of processors, N, is increased. This raises the question of how to choose N to maximize speed and keep accuracy high. The approach discussed here uses a threshold of goodness for rules. Rules that perform below the threshold are deleted from the final rule set. The question of how to most effectively set the threshold is an open one.

It is shown that the performance of rules can be improved by further specializing those that are under performing. Conditions can be added as the rules are applied tested on data stored on other processors. Both the specialized rules and original rules remain as long as their accuracy is above the threshold.

We have pointed out issues and potential fixes to an approach that promises to provide scalable, accurate rules generated from a parallel computing system. It will enable learning from large distributed data sets.

Acknowledgments

This research was partially supported by the United States Department of Energy through the Sandia National Laboratories LDRD program, contract number DE-AC04-76DO00789

References

[Chan and Stolfo, 1996] P.K. Chan and S.J. Stolfo. Scaling learning by meta-learning over disjoint and partially replicated data. In *Proceedings of the Florida Artificial Intelligence Society*, 1996.

[Chan and Stolfo, 1998] P. K. Chan and S. J. Stolfo. Toward scalable learning with non-uniform class and cost distributions: A case study in credit card fraud detection. In *Proc. KDD-98*, 1998.

[Clearwater *et al.*, 1989] S. Clearwater, T. Cheng, H. Hirsh, and B. Buchanan. Incremental batch learning. In *Proceedings of the Sixth Int. Workshop on Machine Learning*, pages 366–370, 1989.

[Cohen, 1995] W.W. Cohen. Fast effective rule induction. In *Proceedings of the 12th Conference on Machine Learning*, 1995.

[Fisher, 1936] R.A. Fisher. The use of multiple measurements in taxonomic problems. *Ann. Eugenics*, 7, 1936.

[Hall *et al.*, 1998] L.O. Hall, N. Chawla, and K.W. Bowyer. Decision tree learning on very large data sets. In *International Conference on Systems, Man and Cybernetics*, pages 2579–2584, Oct 1998.

[Kufrin, 1997] R. Kufrin. Generating C4.5 production rules in parallel. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 565–570, July 1997.

[Merz and Murphy,] C.J. Merz and P.M. Murphy. *UCI Repository of Machine Learning Databases*. Univ. of CA., Dept. of CIS, Irvine, CA. http://www.ics.uci.edu/~mlearn/MLRepository.html.

[Provost and Hennessy, 1994] F.J. Provost and D. Hennessy. Distributed machine learning: Scaling up with coarse-grained parallelism. In *Proceedings of the Second International Conference on Intelligent Systems for Molecular Biology*, 1994.

[Provost and Hennessy, 1996] F.J. Provost and D.N. Hennessy. Scaling up: Distributed machine learning with cooperation. In *Proceedings of AAAI'96*, pages 74–79, 1996.

[Quinlan, 1987] J.R. Quinlan. Generating production rules from decision trees. In *Proceedings of IJCAI-87*, pages 304–307, 1987.

[Quinlan, 1992] J.R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1992. San Mateo, CA.

[Quinlan, 1996] J.R. Quinlan. Improved use of continuous attributes in C4.5. *Journal of Artificial Intelligence Research*, 4:77–90, 1996.

[Stolfo *et al.*, 1997] S.J. Stolfo, A.L. Prodromidis, S. Tselepis, W. Lee, D. Fan, and P.K. Chan. JAM: Java agents for meta-learning over distributed databases. In *Proc. KDD-97*, 1997.

[Weiss *et al.*, 1990] S.M. Weiss, R.S. Galen, and P.V. Tadepalli. Maximizing the predictive value of production rules. *Artificial Intelligence*, 45:47–71, 1990.

[Williams, 1990] G.J. Williams. *Inducing and Combining Multiple Decision Trees*. PhD thesis, Australian National University, Canberra, Australia, 1990.