# Towards a Cost-Effective Parallel Data Mining Approach

Zoltán Járai, Aashu Virmani, Liviu Iftode
Department of Computer Science
Rutgers University
Piscataway, NJ 08854
{jarai,avirmani}@paul.rutgers.edu, iftode@cs.rutgers.edu

## Abstract

*Massive rule induction has recently emerged as one of the powerful data mining techniques. The problem is known to be exponential in the size of the attributes, and given its ever increasing use, can greatly benefit from parallelization.*

*In this paper, we study cost-effective approaches to parallelize rule generation algorithms. In particular, we consider the propositional rule generation algorithm of the Discovery Board system, and present our design and implementation of a parallel algorithm for the same task. We then present some early performance results of our parallelization scheme on hardware and software distributed shared memory multiprocessors.*

## 1 Introduction

Massive rule induction has recently emerged as one of the powerful data mining techniques. However, the technique is well known to suffer from a combinatorial explosion, and the number of rules generated can often be an order of magnitude larger than data itself. The algorithms for this problem must explore all combinations of attribute-values (of all lengths), with several such sets of attribute combinations being mutually "independent." The problem thus lends itself naturally to parallelization, and can greatly benefit from it.

Much of the early work done in this area [1, 2, 9, 4] has dealt with one particular type of problem: market basket analysis. This problem deals with generation of association rules when the data consists of boolean valued attributes present within a transaction. The problem is generally known to be exponential in the number of attributes in the data. An enhancement of this approach involves generating propositional rules (also called generalized association rules) from data where the attributes can contain arbitrary discrete and continuous values.

One approach to propositional rule generation involves trivially "flattening" an attribute $A$ with $k$ values into $k$ boolean attributes, each being a predicate of the form $(A = a_i)$. However, this approach further multiplies the number of attributes, increasing the problem size substantially. The the DISCOVERY BOARD system [5] generates such generalized association rules directly from relational data, which can be persistently stored and later filtered using rule-queries. We attempt to parallelize this algorithm, and present our observations and results in this paper.

The goal of our research is cost-effective parallelization of data mining algorithms. Hardware-coherent shared memory systems usually provide very good performance but are expensive. Given the growing need for data mining and the increasing number of huge data sets, a more cost-effective solution is called for. As an alternative to hardware shared memory platforms, software distributed shared memory systems [7] represent an inexpensive solution to provide the shared address space abstraction on any network of computers. Recent results also indicate that the performance of software distributed shared memory systems improved substantially by using efficient protocols [10], low-latency communication architecture [3], and adequate parallel application structuring [6].

As a first step towards obtaining this goal we designed a parallel algorithm and we are starting to evaluate it on a hardware shared memory system. This would not only yield us preliminary estimation of the effectiveness of the algorithm but also provide us with an upper bound on the performance of software DSM systems.

The rest of the paper is organized as follows. We briefly present some concepts and the sequential algorithm. Next, we describe the approach we have taken in parallelizing it, and discuss our data structures and the algorithm. Finally, we present results of the sequential implementation and the implementation plan, under which we propose to test and measure the parallel algorithm.

## 2 Background

Let $R$ be a relation defined on the set of attributes $A_1, A_2, ...A_m$. By a *descriptor* we mean a pair (*attribute*, *value*). To *satisfy* a descriptor $(A, a)$, a data tuple must have the value *a* for attribute *A*. For continuous valued attributes, the *value* could be a pair (*lo*, *hi*). In this case, descriptor $(A, a)$ satisfies a conjunct if $lo \leq a \leq hi$ with respect to a given ordering. A *conjunct* is a set of descriptors, such that each descriptor within a conjunct has a different attribute. If we think of a descriptor as a condition $(A = a)$ on the data, then a conjunct represents a logical AND of the conditions represented by its component descriptors. The set of attributes appearing in the descriptors of a conjunct is called the *signature* of a conjunct.

Each conjunct $C$ is associated with an inverted list—a list of all tuple identifiers $T$, which satisfy it. The pair $(C, T)$ is called a *C-list* associated with conjunct $C$. If conjunct $C$ has a single descriptor, that is $C = \{d\}$, the associated pair $(d, T)$ is called a *d-list*. Additionally, the count of all tuples satisfying $C$ is called the *support* of the conjunct.

A *propositional rule* over $R$ is an implication of the form:

$$\textit{Body} \Longrightarrow \textit{Consequent}$$

where *Body* is a conjunct and *Consequent* is a single descriptor. Additionally, each rule is specified by two parameters: *support* and *confidence*. The **support** of the rule is defined as the number of tuples satisfying the body of the rule. The **confidence** of a rule is defined as the ratio of the number of all tuples which satisfy both the body and the consequent to the number of all tuples which satisfy just the body of the rule. Furthermore, the **length** of a rule is the number of descriptors in its body.

An example of such a rule might be:

$(\textit{location} = \text{Eastern})$ **and** $(\textit{percapita} \in [22000\text{–}30000])$ **and**
$(\textit{prez\_92} = \text{Democrat}) \Longrightarrow (\textit{gov\_94} = \text{Republican}) \quad [83\%, 6]$

The above rule can be read as: *"Eighty-three percent of the 6 eastern states with per capita income between £22,000 and £30,000 who voted for a Democratic president in 1992, voted for a Republican governor in 1994."*

**The Problem:** The task of the massive rule induction process is to generate all such generalized association rules which lie above the given support and confidence thresholds.

# 3 Sequential Algorithm

The sequential algorithm we intend to parallelize is the breadth-first algorithm of the DISCOVERY BOARD system. The algorithm works level-wise, generating all rules of length $i$ before generating the first rule of length $i + 1$.

The actual step of rule generation follows the expansion of a conjunct of length $k$ to a conjunct of length $k + 1$.

The C-list can be viewed as an *index* or an *inverted list* using standard database terminology. The algorithm follows an *index-ANDing* approach, where C-lists for significant conjuncts are merged to generate higher length conjuncts. This is illustrated in figure 1 below.

**C-list Extension and C-list Expansion:** A C-list $(C, T_1)$ can be **extended** by the d-list $(d, T_2)$ where $d = (a, v)$ iff $a$ does not occur in $C$ and is more senior (with respect to attribute ordering) than any attribute that occurs in the descriptors in $C$. The **expansion** of the C-list, on the other hand, can occur with *any* d-list $d = (a, v)$ such that $a$ does not occur in $C$. The result of both the extension and the expansion is a new C-list $(C \cup d, T_1 \cap T_2)$.

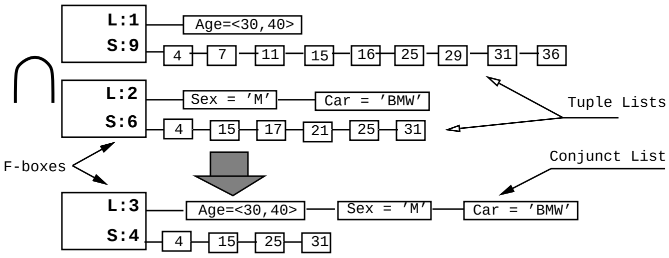



**Figure 1. Merging two C-lists.**

For purposes of the algorithm, we will assume that attributes of the original database are linearly ordered. We define a structure called a *frontier list* as follows: frontier list of length $i$, ($FL_i$ in short), contains all C-lists of length $i + 1$ that meet the minimum support. The first frontier list, $FL_0$ is constructed by making one pass through the database. The rest of the algorithm follows a uniform "extend-by-one" approach, combining $FL_i$ with $FL_0$ to get $FL_{i+1}$. During rule generation for a new conjunct in $FL_{i+1}$, we look for all immediate predecessors of the conjunct in $FL_i$, and generate a rule by examining their respective supports. The sequential algorithm is outlined in Figure 2.

**main():**
  **for** $i = 1 \ldots \textit{attribute\_count} - 1$ **do**
    $FL_{i+1}$ = *extend_by_one*($FL_i$, $FL_0$);
    *destroy*($FL_i$);

**extend_by_one($FL_i$, $FL_0$):**
  **for** each C-list $CL$ in $FL_i$
    extend $CL$ with with all "senior" descriptors in $FL_0$
    if resulting C-list $CL_{new}$ meets *min_support*
      add $CL_{new}$ to $FL_{i+1}$;
      *generate_all_rules*($CL_{new}$);

**generate_all_rules($CL = (C, L)$):**
  **for** each descriptor $d \in C$
    find support of conjunct $(C - d)$ from $FL_{len-1}$
    generate rule $(C - d) \Longrightarrow d$ if it meets confidence

**Figure 2. The sequential algorithm.**

# 4 Design for Parallelization

## 4.1 Parallel Approaches

The sequential algorithm proceeds in an iterative manner repeatedly executing the algorithm *extend_by_one*. Each step heavily relies on the previous one and only on the previous one: for every conjunct of $FL_{i+1}$ created, several lookups in $FL_i$ are required. Thus a natural way to parallelize the algorithm is to parallelize each step. Since the creation of a single conjunct cannot be parallelized, tasks of creating conjuncts have to be assigned to processors. The question is then: how do we partition $FL_i$ into groups of conjuncts and assign conjunct and rule generation tasks based on the partitioning? There are several possible approaches:

1. **Partition by entities of database.** We may divide the original data without any knowledge of data values and then create conjuncts from them. For example, we might partition the database into chunks of records. This approach offers good load balancing due to static partitioning but determining the real support of a conjunct becomes cumbersome and results in high communication. Another way is to partition the database such that each task consists of an attribute. Then determining the support is no longer a problem but combining results for longer conjuncts yields in heavy communication traffic.

2. **Partition by value.** To overcome large communication overhead we could partition the data according to values of attributes, that is, each task gets part of the database having $A = a_i$ where $A$ is an attribute and $a_i$ is a possible value of $A$. Load balancing is a serious problem here. Also, considering conjuncts which do not contain $A$, one of the groups is the "$A$ is missing" branch, which is effectively a large part.

3. **Partition conjuncts of $FL_i$.** Another idea is to partition every $FL_i$ independently but compute $FL_0$ sequentially. First, we might let the creation of every single conjunct be a separate tasks but then tasks become too small which results in too much processor communication. Also, the critical section of getting a new task is entered very frequently. To make tasks larger the next step is to let the creation of several conjuncts be a task. How should we choose those groups of conjuncts? The groups should not be too large to ensure good load balancing. The idea is **signature based partitioning**. We partition $FL_i$ such that conjuncts having the same signature constitute a group. This way the size of tasks becomes bigger but still manageable. We extend the meaning of signature by calling the group itself a *signature*. Our parallel algorithm is based on this approach.

## 4.2 Overall Algorithm

The parallel algorithm proceeds in breadth first mode, expanding $FL_i$ by $FL_0$. Each of these expansions has two steps: a *conjunct extension* phase for extending $FL_i$ by $FL_0$ to get $FL_{i+1}$ and a *rule generation* phase to generate rules of length $i$. The conjunct generation phase **extends** existing conjuncts to avoid regeneration of the target conjunct, since expanding $AC$, $AD$, or $CD$ all could yield $ACD$ but only extending $AC$ yields $ACD$. Rule generation, on the other hand considers all **expansions** of a conjunct to get all rules, since while a conjunct like $ACD$ is essentially unordered, the three rules $AC \Rightarrow D$, $AD \Rightarrow C$, and $CD \Rightarrow A$ are different, and must all be separately enumerated.

During both of these phases each processor executes tasks of the same kind. Employing signature based task division each task is the extension or expansion of a signature of $FL_i$ by an attribute from $FL_0$. A task is represented by a quadruple

$$(\textit{incarnation}, \textit{weighted sum}, \textit{signature}, \textit{attribute}),$$

where *incarnation* is a number denoting the $FL$ to be expanded, *signature* stands for a group of conjuncts having the same signature, and *attribute* is a single attribute in $FL_0$. An estimate proportional to the amount of work to be done on the task is given by *weighted sum* which is used for load balancing when distributing tasks among task queues as we shall see later. Note the twofold nature of these quadruples: during the conjunct extension phase they stand for the *extension* of a signature by attribute while in the rule generation phase they mean the generation of all rules with *signature* as their body, and *attribute* as their consequent. For instance, the task $(i, w, ABD, E)$ in the conjunct generation phase represents extending all conjuncts of the form $(A = *)(B = *)(D = *)$, with all descriptors of the form $(E = *)$. In the rule generation phase, the task involves generating all rules of the form $(A = *)(B = *)(D = *) \Longrightarrow (E = *)$.

## 4.3 Tasks

Two kinds of tasks are used: *conjunct tasks* and *rule tasks*. During the conjunct extension phase of incarnation $i$, tasks from the *conjunct task queue*, $CTQ_i$ are executed, new tasks for the next incarnation are inserted into $CTQ_{i+1}$, and tasks for making rules of length $i$ are put into the *rule task queue*, $RTQ_i$. For each conjunct of length $i$ built, all rule generation tasks with that left-hand side are created. As soon as all new conjuncts are created, processors enter the rule generation phase and process tasks from $RTQ_i$. After all rules are generated, incarnation $i+1$ follows. Since at this stage all rules of length $i$ have been generated, conjuncts of length $i$ are no longer needed, and $CTQ_i$ and $RTQ_i$ are destroyed. (Thus, effectively, only three tasks queues are needed.) Task queues are shown in Figure 3.

Redistribution is used to achieve load balance and minimize communication. Tasks are redistributed before each phase based on their weight and, to improve load balance processors "steal" tasks from others after finishing their own. The moderate size of tasks, due to the size of signatures, helps us achieve good load balance. Since creating rules of length $i$ requires lookups in only $i + 1$ signatures of $FL_i$ (see Figure 4), we can maintain locality by assigning tasks which do lookups in the same signatures of length $i$ to the same processor. But also the major problem arises from the overlapping of tasks (see figure 5). Usually a high

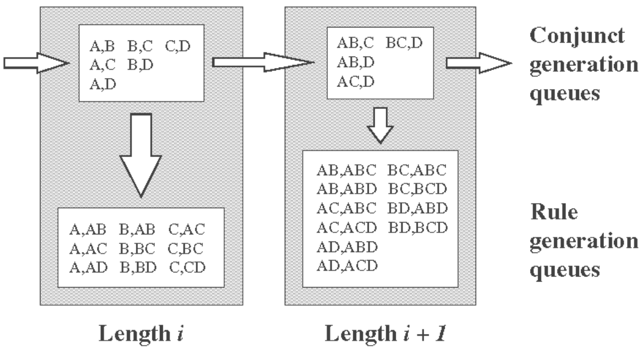



**Figure 3. Task Queues.**

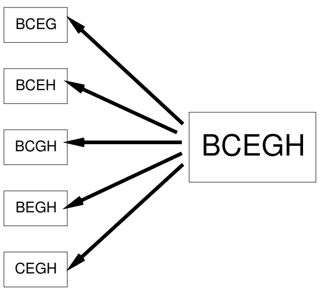



**Figure 4. Creation of a conjunct in $FL_{i+1}$ requires $i+1$ lookups in $FL_i$.**

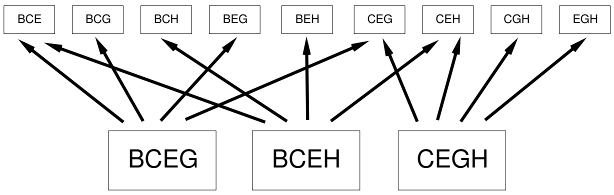



**Figure 5. Lookups in $FL_{i+1}$ can overlap when creating rule of length $i$.**

number of overlapping occurs and hence we cannot expect all overlapping tasks to be assigned to the same processor. Thus the overlaps result in data replication. To minimize data replication we must distribute signatures of $FL_i$ in a way that the signatures assigned to a processor can contribute to a maximum number of rules of length $i$.

**Idea for redistribution.** When redistributing tasks we would like to make sure that (1) each processor will be equally loaded and (2) minimal amount of data will be replicated. One approach is the following: create a directed graph where each vertex represents a signature $s$ in $FL_{i+1}$ and stores the estimated time to create $s$ and the time to get the required signature lookup tables of $FL_i$. An arc pointing from vertex $u$ to $v$ specifies the amount of time to load signature lookup tables required by $v$ additionally to those of $u$. Then perform the following algorithm: initially each vertex forms a group of its own. At every iteration the least weighted arc $a$ leaving the least weighted group $g$ is selected and $g$ is merged with $h$, the group pointed to by $a$. We repeat this process until we have exactly as many groups as processors. Since $g$ itself is small and $a$ is also minimal, group $h$ cannot bring a huge amount of work. After joining the two groups, we update the arcs leaving $g$ and $h$. The algorithm could be applied to dynamic networks by doing a new iteration only when a new task is needed or by re-executing the algorithm.

## 5 Parallel Implementation Plan

### 5.1 Parallel Platforms

To evaluate the performance space of our parallel data mining scheme we plan to implement and evaluate the algorithm on two platforms: a hardware shared memory multiprocessor and a software shared memory system.

To evaluate the performance potential of our parallel algorithms we will first implement them on a shared memory multiprocessor which supports parallel processing at a very low inter-processor communication cost. We chose a 16-node SGI Challenge which is a bus-based, symmetric shared memory multiprocessor with centralized memory. Processor's speed is 150 MHz and the total size of the main memory is 1 Gbyte.

The primary goal of our research is to demonstrate that the software shared memory approach [7] can be a cost-effective parallel platform for data mining. We chose a home-based lazy-release consistency protocol (HLRC) [10] which was shown to provide good performance and scalability. For our experiments we plan to implement a multi-threaded HLRC on an 8-node SMP cluster based on Myrinet interconnect. Each node has two 300 MHz Pentium II processors with a 128 MBytes of memory which brings the total main memory available on the cluster to 1 Gbyte.

The data mining programs are written in C using the the Argonne National Laboratories parmacs [8] macros for parallel constructs. Since both parallel platforms support this

API programming we expect to our programs to be portable between these two environments.

## 5.2 Current Status

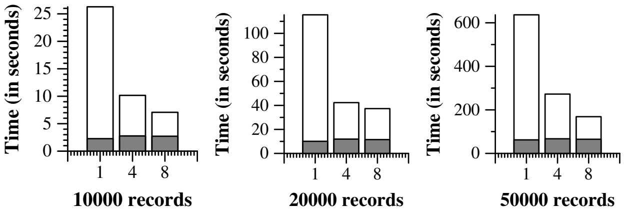
At this point, we are able to test a parallel version of our algorithm on hardware shared memory platform only. We are in the process of fine-tuning the implementation to remove some of the synchronization bottlenecks before we test the same implementation on a software shared memory platform. The machine used was a 16 node SGI Challenge as described in section 5.1. Of the 1GB of RAM available, 500MB was allocated for this task. For now, we kept the problem sizes small enough to fit in the memory, to factor out variations due to paging and swapping.

We performed two sets of experiments to get a basic idea of the program's behavior. The first experiment involved using a fixed number of attributes (8), and varying the data sizes between 10,000 and 50,000 records, using 1, 4 and 8 processors respectively. Figure 6 shows the results from this experiment. The second experiment, shown in figure 7 studied the growth in time and speedups obtained for a fixed data size (100,000 records), but increasing number of columns from 6 to 10.

The synthetic data was generated using the following method: The input parameters to the program were the number of attributes $n$, the number of records $m$, the minimum support $s$, and a correlation-distribution $D$. The correlation distribution specified for all lengths between 0 and $n$, the percentage of attributes which were to be correlated. In other words, if $D(3)$ was 25%, it meant that 25% of the data was correlated in exactly 3 attributes. (Note that a $D(4)$ indirectly contributes to $D(3)$, $D(2)$ and $D(1)$ also). Data was generated using clusters (a set of contiguous rows) of size $s$ – the minimum support. We divided $n$ by $s$ to determine the number of clusters, and for each cluster we picked a number $k$ and made $k$ of the columns (randomly chosen) correlated, while filling the others with unique values, so as to avoid spurious correlations. Such a cluster was called $k$-*correlated*. The number $k$ was chosen based on the overall distribution, such that we ended up with $D(i)$ percent of $i$-correlations. The minimum support was set to be $s$, and the confidence was set to be 1.0. The program thus allowed us to precisely control the correlation level of the data, enabling us to study the performance over increasing data sizes (in terms of attributes or records) of similar nature.

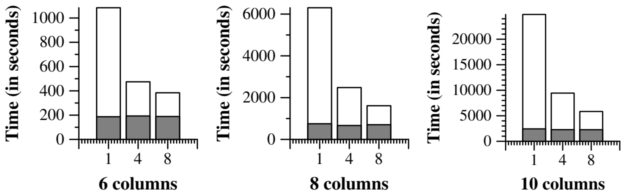
**Observations:** Increasing the problem size seems to increase the benefit of parallelization, perhaps because the percentage of overhead costs diminishes. We profiled several portions of the parallel version, and the two most prominent tasks within the algorithm turn out to be those of conjunct generation and rule generation. The shaded area in the graphs represents the conjunct generation phase and the unshaded part represents the time spent on rule generation.

The conjunct generation portion, at this point, seems to suffer from a memory contention problem, and is thus




| numRows | number of processors | | |
|---|---|---|---|
| | 1 | 4 | 8 |
| 10,000 | 1 | 1.67 | 1.85 |
| 20,000 | 1 | 1.99 | 2.10 |
| 50,000 | 1 | 1.94 | 2.52 |

**Figure 6. Speedup for varying number of processors for increasing data sizes (number of rows)**




| numCols | number of processors | | |
|---|---|---|---|
| | 1 | 4 | 8 |
| 6 | 1 | 1.86 | 2.04 |
| 8 | 1 | 2.26 | 3.14 |
| 10 | 1 | 2.49 | 3.63 |

**Figure 7. Speedup for varying number of processors for increasing data sizes (number of columns)**

achieving little or no reduction as a result of increasing the number of processors. Once we manage to correct this aberration, we believe that we can get close to a speedup of around 6 when using 8 processors. We hope to present more detailed results of our experiments, and comparisons with software shared memory platform in the near future.

To cope with the very long execution times common to data mining applications we are particularly interested in extending the software distributed shared memory protocol to support thread migration, dynamic reconfiguration and fault tolerance. In a reconfigurable cluster setup nodes are allowed to join the cluster dynamically as they become available and to retire when they become loaded with interactive tasks.

## 6 Conclusions

The increasing importance of data mining applications requires algorithms and platforms that enable manipulating huge data sets in a fast and cost-efficient manner. This paper is a first step towards reaching this goal. Based on our sequential algorithm we presented the design and implementation of a shared memory parallelization approach, along with some preliminary results. As part of continuing work, we plan to fine tune and test this implementation on a software distributed shared memory system.

## References


[1] R. Agrawal, T. Imielinski, and A. Swami. Mining associations rules between sets of items in large databases. In *Proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD'93)*, pages 207 – 216, Washington D.C., May 1993.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *vldb94*, pages 487 – 499, Santiago, Chile, 1994.

[3] A. Bilas and J. P. Singh. The effects of communication parameters on end performance of shared virtual memory clusters. In *Proceedings of the 1997 SC97 Conference*, San Jose, CA, Nov. 1997.

[4] S. Brin, R. Motwani, J. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD'97)*, pages 255 – 264, Tuscon, Arizona, May 1997.

[5] T. Imielinski, A. Virmani, and A. Abdulghani. Datamine: Application programming interface and query language for database mining. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96)*, Portland, Oregon, August 1996.

[6] D. Jiang, H. Shan, and J. P. Singh. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming*, June 1997.

[7] K. Li. *Shared Virtual Memory on Loosely-coupled Multiprocessors*. PhD thesis, Yale University, Oct. 1986. Tech Report YALEU-RR-492.

[8] E. L. Lusk and R. A. Overbeek. A minimalist approach to portable, parallel programming. In L. H. Jamieson, D. B. Gannon, and R. J. Douglass, editors, *The Characteristics of Parallel Algorithm*, pages 351–362. MIT Press, 1987.

[9] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB'95)*, pages 432 – 444, Zurich, Switzerland, Sept 1995.

[10] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *Proceedings of the Operating Systems Design and Implementation Symposium*, Oct. 1996.