# New Parallel Algorithms for Frequent Itemset Mining in Very Large Databases

Adriano Veloso and Wagner Meira Jr.
Computer Science Dept.
Universidade Federal de Minas Gerais
{adrianov, meira}@dcc.ufmg.br

Srinivasan Parthasarathy
Computer and Information Science Dept.
The Ohio-State University
srini@cis.ohio-state.edu

## Abstract

*Frequent itemset mining is a classic problem in data mining. It is a non-supervised process which concerns in finding frequent patterns (or itemsets) hidden in large volumes of data in order to produce compact summaries or models of the database. These models are typically used to generate association rules, but recently they have also been used in far reaching domains like e-commerce and bio-informatics. Because databases are increasing in terms of both dimension (number of attributes) and size (number of records), one of the main issues in a frequent itemset mining algorithm is the ability to analyze very large databases. Sequential algorithms do not have this ability, especially in terms of run-time performance, for such very large databases. Therefore, we must rely on high performance parallel and distributed computing. We present new parallel algorithms for frequent itemset mining. Their efficiency is proven through a series of experiments on different parallel environments, that range from shared-memory multiprocessors machines to a set of SMP clusters connected together through a high speed network.*

## 1. Introduction

Our ability to collect and store data is fair outpacing our ability to analyze it. This phenomenon is mainly explained by the gap between advances in computing power and storage capacity technologies. Moore's law states that computing power doubles approximately every 18 months, while the corresponding law for advances in storage capacity technology is even more impressive – storage capacity doubles approximately every 9 months [5]. As a consequence, the value of data is no longer in "how much of it you have" or neither in "how fast can you gather it", but in how quickly and how effectively can the data be reduced and explored. Thus, implementation of non-trivial data mining ideas in high performance parallel and distributed computing environments is becoming crucial. Clearly, such parallel computing environments greatly increase our ability to analyze data, since the computing power is increased. However, these environments also pose several challenges to data mining, such as high communication and synchronization overhead, data skewness, and workload balancing.

In this paper we present new parallel algorithms for a key data mining task: the discovery of frequent itemsets. This task has a simple problem statement: to find the set of all subsets of items (or attributes) that frequently occur together in database transactions (or records). Although the frequent itemset mining task has a simple statement, it is both CPU and I/O intensive, mostly because of the high dimension and large size of the databases involved in the process. Several efficient sequential algorithms were proposed in the literature [2, 12, 9]. There are also some parallel algorithms [1, 7, 6, 13, 10]. However, these algorithms need several rounds of communication, incurring in serious synchronization and I/O overhead. Our new algorithms need only one round of communication while searching for the frequent itemsets and one reduction operation to generate the correct model. Further, they can make use of both shared- and distributed-memory advantages and deal with both high dimension and large size problems. We evaluated our algorithms through a broad series of experiments using different parallel environments.

## 2. Definitions and Preliminaries

DEFINITION 1. [ITEMSETS] For any set $\mathcal{X}$, its size is the number of elements in $\mathcal{X}$. Let $\mathcal{I}$ denote the set of $n$ natural numbers $\{1, 2, ..., n\}$. Each $x \in \mathcal{I}$ is called an item. A non-empty subset of $\mathcal{I}$ is called an itemset. The power set of $\mathcal{I}$, denoted by $\mathcal{P}(\mathcal{I})$, is the set of all possible subsets of $\mathcal{I}$. An itemset of size $k$, $\mathcal{X} = \{x_1, x_2, ..., x_k\}$ is called a $k$-itemset (for convenience we drop set notation and denote $\mathcal{X}$ as $x_1x_2...x_k$). For $\mathcal{X}, \mathcal{Y} \in \mathcal{P}(\mathcal{I})$ we say that $\mathcal{X}$ contains $\mathcal{Y}$ if $\mathcal{Y} \subseteq \mathcal{X}$. A set (of itemsets) $\mathcal{C} \subseteq \mathcal{P}(\mathcal{I})$ is called an itemset collection, and $\mathcal{C}$ is a *Sperner* collection if no itemset in it contains another: $\mathcal{X}, \mathcal{Y} \in \mathcal{C}$, and $\mathcal{X} \neq \mathcal{Y}$ implies $\mathcal{X} \nsubseteq \mathcal{Y}$.

DEFINITION 2. [TRANSACTIONS] A transaction $\mathcal{T}_i$ is an itemset, where $i$ is a natural number called the transaction identifier or $tid$. A transaction database $\mathcal{D} = \{\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_m\}$, is a finite set of transactions, with size $|\mathcal{D}| = m$. The absolute support of an itemset $\mathcal{X}$ in $\mathcal{D}$ is the number of transactions in $\mathcal{D}$ that contain $\mathcal{X}$, given as $\pi(\mathcal{X}, \mathcal{D}) = |\{\mathcal{T}_i \in \mathcal{D} \,|\mathcal{X} \subseteq \mathcal{T}_i\}|$. The (relative) support of an itemset $\mathcal{X}$ in $\mathcal{D}$ is the fraction of transactions in $\mathcal{D}$ that contain $\mathcal{X}$, given as, $\sigma(\mathcal{X}, \mathcal{D}) = \frac{\pi(\mathcal{X},\mathcal{D})}{|\mathcal{D}|}$.

DEFINITION 3. [FREQUENT AND MAXIMAL FREQUENT ITEMSETS] An itemset $\mathcal{X}$ is frequent if $\sigma(\mathcal{X}, \mathcal{D}) \geq \sigma^{min}$, where $\sigma^{min}$ is a user-specified minimum-support threshold, with $0 < \sigma^{min} \leq 1$. A collection of frequent itemsets is denoted as $\mathcal{F}(\sigma^{min}, \mathcal{D})$. A frequent itemset $\mathcal{X} \in \mathcal{F}(\sigma^{min}, \mathcal{D})$ is $maximal$ if it has no frequent superset. A collection of maximal frequent itemsets is denoted as $\mathcal{MF}(\sigma^{min}, \mathcal{D})$. $\mathcal{MF}(\sigma^{min}, \mathcal{D})$ is a $Sperner$ collection on $\mathcal{I}$.

LEMMA 1. [2] Any subset of a frequent itemset is frequent: $\mathcal{X} \in \mathcal{F}(\sigma^{min}, \mathcal{D})$ and $\mathcal{Y} \subseteq \mathcal{X}$ implies $\mathcal{Y} \in \mathcal{F}(\sigma^{min}, \mathcal{D})$. Thus, by definition a frequent itemset must be subset of at least one maximal frequent itemset. ■

LEMMA 2. [12] $\mathcal{MF}(\sigma^{min}, \mathcal{D})$ is the smallest collection of itemsets from which $\mathcal{F}(\sigma^{min}, \mathcal{D})$ can be inferred. ■

PROBLEM 1. [MINING FREQUENT ITEMSETS] Given $\sigma^{min}$ and a transaction database $\mathcal{D}$, the problem of mining frequent itemsets is to find $\mathcal{F}(\sigma^{min}, \mathcal{D})$.

EXAMPLE 1. Let us consider the example in Figure 1, where $\mathcal{I}$ = {1,2,3,4,5} and the figure shows $\mathcal{D}$ and $\mathcal{P}(\mathcal{I})$. Suppose $\sigma^{min}$ = 0.4 (40%). $\mathcal{F}(0.4, \mathcal{D})$ is composed by the shaded and bold itemsets, while $\mathcal{MF}(0.4, \mathcal{D})$ is composed only by the bold itemsets. Note that $|\mathcal{MF}(0.4, \mathcal{D})|$ is much smaller than $|\mathcal{F}(0.4, \mathcal{D})|$.

A naive approach to find $\mathcal{F}(0.4, \mathcal{D})$ is to first compute $\sigma(\mathcal{X}, \mathcal{D})$ for each $\mathcal{X} \in \mathcal{P}(\mathcal{I})$, and then return only those that $\sigma(\mathcal{X}, \mathcal{D}) \geq 0.4$. This approach is inappropriate because:

- If $|\ \mathcal{I}\ |$ (the dimension) is high, then $|\ \mathcal{P}(I)\ |$ is huge (i.e., $2^{|\mathcal{I}|}$).
- If $|\ \mathcal{D}\ |$ (the size) is large, then computing $\sigma(\mathcal{X}, \mathcal{D})$ for all $\mathcal{X} \in \mathcal{P}(\mathcal{I})$ is infeasible.

By applying lemma 1 we can greatly improve the search for $\mathcal{F}(0.4, \mathcal{D})$. In Figure 1, once we know that the itemset {34} is not frequent, we do not need to generate any of its supersets, since they must also be not frequent. This simple pruning trick was first used in [2] and it greatly reduces the number of candidate itemsets generated. Several different searches can be applied by using this pruning trick.

*Basic Algorithm:* Our algorithm employs a backtrack search to find $\mathcal{F}(\sigma^{min}, \mathcal{D})$ and $\mathcal{MF}(\sigma^{min}, \mathcal{D})$. A solution is represented as an itemset $\mathcal{X} = \{x_0, x_1, ...\}$. Each

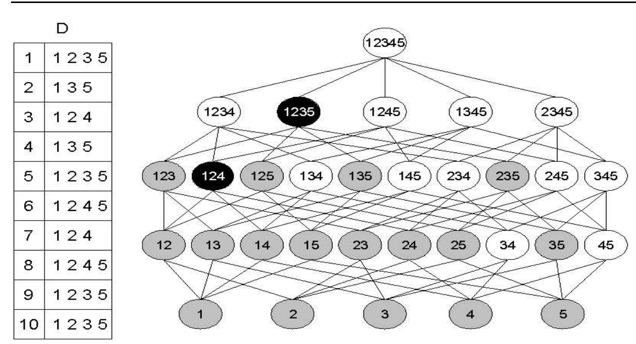


**Figure 1. Frequent Itemset Mining Example.**

item $x_j$ is chosen from a finite *possible set*, $\mathcal{P}_j$. Initially $\mathcal{X}$ is empty; it is extended one item at a time, as the search proceeds. The size of $\mathcal{X}$ is the same as the depth of the corresponding itemset in the search tree. Given a $k$-candidate itemset $\mathcal{X}_k = \{x_0, x_1, ..., x_{k-1}\}$, the possible values for the next item $x_k$ comes from a subset $\mathcal{R}_k \subseteq \mathcal{P}_k$ called the *combine set*. Each iteration tries extending $\mathcal{X}_k$ with every item $x$ in $\mathcal{R}_k$. An extension is valid if the resulting itemset $\mathcal{X}_{k+1}$ is frequent. If $\mathcal{X}_{k+1}$ is frequent and it is no subset of any already known maximal frequent itemset, then $\mathcal{X}_{k+1}$ is a maximal frequent itemset. The next step is to extract the new possible set of extensions, $\mathcal{P}_{k+1}$, which consists only of elements in $\mathcal{R}_k$ that follow $x$. The new combine set, $\mathcal{R}_{k+1}$ consists of those elements in the possible set that produce a frequent itemset when used to extend $\mathcal{X}_{k+1}$. Any item not in the combine set refers to a pruned subtree. The algorithm performs a depth-first traversal of the search space. When the search finishes, $\mathcal{F}(\sigma^{min}, \mathcal{D})$ and $\mathcal{MF}(\sigma^{min}, \mathcal{D})$ were found. The computation of $\sigma(\mathcal{X}, \mathcal{D})$ is based on the associativity of subsets. Let $\mathcal{L}_{\mathcal{D}}(\mathcal{X})$ be the $tidset$ of $\mathcal{X}$ in $\mathcal{D}$ (the set of $tids$ in $\mathcal{D}$ in which $\mathcal{X}$ has occurred), and thus, $|\ \mathcal{L}_{\mathcal{D}}(\mathcal{X})\ | = \pi(\mathcal{X}, \mathcal{D})$. According to [12], $\mathcal{L}_{\mathcal{D}}(\mathcal{X})$ can be obtained by intersecting the tidsets of two subsets of $\mathcal{X}$. For example, in Figure 1, $\mathcal{L}_{\mathcal{D}}(123)$ = {1, 5, 9, 10} and $\mathcal{L}_{\mathcal{D}}(125)$ = {1, 4, 5, 8, 9, 10}. Consequently, $\mathcal{L}_{\mathcal{D}}(1235) = \mathcal{L}_{\mathcal{D}}(123) \cap \mathcal{L}_{\mathcal{D}}(125)$ = {1, 5, 9, 10}. $|\mathcal{L}_{\mathcal{D}}(1235)| = \pi(1235, \mathcal{D}) = 4$, and thus $\sigma(1235, \mathcal{D}) = 0.4$.

EXAMPLE 2. Figure 2 shows how the algorithm works. We used sequence numbers above each itemset to facilitate the understanding of the backtrack search. First the algorithm process the items, and then it starts a depth-first search for frequent itemsets. For example, for sequence number $s$ = 13, the itemset being processed is {124}. Therefore, the depth is $k = 3$, and $\mathcal{P}_3 = \mathcal{R}_3 = \{5\}$. When $s$ = 15, the algorithm visits the itemset {124} again, but now $\mathcal{R}_3 = \emptyset$ and consequently $\{124\} \in \mathcal{MF}(0.4, \mathcal{D})$. Although an itemset $\mathcal{X}$ can be visited more than once, $\sigma(\mathcal{X}, \mathcal{D})$ is computed only in the first visit.

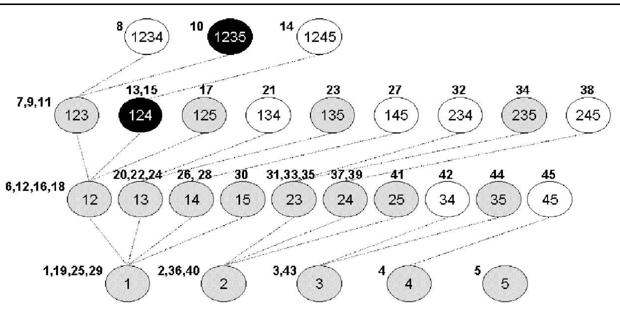


**Figure 2. Basic Algorithm.**

## 3. New Parallel Algorithms

In this section we propose several different parallel algorithms. First we present the design of our parallel approaches, and then we show how to efficiently implement these approaches using different parallel environments.

In our parallel and distributed scenario, $\mathcal{D}$ is divided into $n$ partitions, $\delta_1, \delta_2, ..., \delta_n$. Each partition $\delta_i$ is assigned to a node $\mathcal{N}_i$. A given node $\mathcal{N}_i$ is composed by one or more processor units, one memory unit, and one database $\delta_i$. A set of nodes is called a *cluster*. Our scenario is composed by a set of inter-connected clusters.

### 3.1. Algorithm Design

**3.1.1. The Data Distribution Approach** In this approach all nodes generate the same set of candidate itemsets, and each node $\mathcal{N}_i$ can thus independently get $\sigma(\mathcal{X}, \delta_i)$ for all candidates $\mathcal{X}$. A naive method to get $\sigma(\mathcal{X}, \mathcal{D})$ would be to perform a communication round (and synchronization) between all nodes for each itemset $\mathcal{X}$ generated. Then we could easily check if $\sigma(\mathcal{X}, \mathcal{D}) \geq \sigma^{min}$. This method is obviously inappropriate, since it performs several rounds of communication. Another method would be each node $\mathcal{N}_i$ generating all $\mathcal{X} \in \mathcal{P}(\mathcal{I})$ within its partition $\delta_i$, and then only one sum-reduction operation between all nodes is necessary to get $\sigma(\mathcal{X}, \mathcal{D})\ \forall\ \mathcal{X} \in \mathcal{P}(\mathcal{I})$. This approach is also inappropriate, since processing all $\mathcal{X} \in \mathcal{P}(\mathcal{I})$ is infeasible.

We overcame all these problems and developed an approach which needs only one round of communication and prunes the search space for frequent itemsets (i.e., our approach processes much fewer itemsets than $|\mathcal{P}(\mathcal{I})|$). We first present the basic theoretical foundation of our method.

LEMMA 3. For a given itemset $\mathcal{X}$, if $\sigma(\mathcal{X}, \mathcal{D}) \geq \sigma^{min}$, then $\exists \delta_i \mid \sigma(\mathcal{X}, \delta_i) \geq \sigma^{min}$.

PROOF: If $\sigma(\mathcal{X}, \delta_i) < \sigma^{min} \forall i \in \{1, ..., n\}$, then $\sigma(\mathcal{X}, \mathcal{D}) < \sigma^{min}$, since $\sum_{i=1}^{n} \mid \delta_i \mid = \mid \mathcal{D} \mid$. □

LEMMA 4. $\bigcup_{i=1}^{n} \mathcal{MF}(\sigma^{min}, \delta_i)$ is an upper bound for $\mathcal{MF}(\sigma^{min}, \mathcal{D})$, and therefore it determines all itemsets $\mathcal{X}$ such that $\sigma(\mathcal{X}, \mathcal{D}) \geq \sigma^{min}$.

PROOF: If $\sigma(\mathcal{X}, \mathcal{D}) \geq \sigma^{min}$, then $\mathcal{X}$ must be frequent in at least one partition of $\mathcal{D}$. If $\mathcal{X}$ is frequent in some partition $\delta_l$ ($1 \leq l \leq n$), then it can be inferred by $\mathcal{MF}(\sigma^{min}, \delta_l)$, and consequently by $\bigcup_{i=1}^{n} \mathcal{MF}(\sigma^{min}, \delta_i)$. □

The Data Distribution approach has four distinct phases:

PHASE 1. [FIRST ASYNCHRONOUS PHASE] Each node $\mathcal{N}_i$ reads its local partition $\delta_i$, and constructs the tidsets for all items in $\delta_i$. Then, each node performs our basic algorithm on its partition. At the end of the search each node $\mathcal{N}_i$ will know both $\mathcal{F}(\sigma^{min}, \delta_i)$ and $\mathcal{MF}(\sigma^{min}, \delta_i)$.

PHASE 2. [COMMUNICATION PHASE] Each node $\mathcal{N}_i$ has to exchange its $\mathcal{MF}(\sigma^{min}, \delta_i)$ with every other node, so that at the end of the communication operation, each node can compute $\bigcup_{i=1}^{n} \mathcal{MF}(\sigma^{min}, \delta_i)$. Since only $\mathcal{MF}(\sigma^{min}, \delta_i) \forall i \in \{1, 2, ..., n\}$ is exchanged, the communication overhead is minimized (note that by lemma 2 $\mathcal{MF}(\sigma^{min}, \delta_i)$ is the smallest frequent itemset collection from which $\mathcal{F}(\sigma^{min}, \delta_i)$ can be inferred).

PHASE 3. [SECOND ASYNCHRONOUS PHASE] Now that all nodes know $\bigcup_{i=1}^{n} \mathcal{MF}(\sigma^{min}, \delta_i)$, by lemma 4 they can find $\mathcal{F}(\sigma^{min}, \mathcal{D})$, without further synchronization. Now each node performs a top-down enumeration of itemsets, as follows. Each itemset present in $\bigcup_{i=1}^{n} \mathcal{MF}(\sigma^{min}, \delta_i)$ is broken into $k$ subsets of size $(k-1)$. The support of this itemset can be computed by intersecting the tidsets of each item present in this itemset. This process iterates generating smaller subsets and computing their supports until there are no more subsets to be checked. At the end of this phase, each node will have the same set of candidate itemsets, since the enumeration is based on the same upper bound.

PHASE 4. [REDUCTION PHASE] After all nodes finish the top-down enumeration, a sum-reduction operation is performed to find $\sigma(\mathcal{X}, \mathcal{D})$ for each candidate itemset $\mathcal{X}$ (i.e., $\sigma(\mathcal{X}, \mathcal{D}) = \frac{\sum_{i=1}^{n} \pi(X, \delta_i)}{|\mathcal{D}|}$. Next step is to remove those $\mathcal{X}$ with $\sigma(\mathcal{X}, \mathcal{D}) < \sigma^{min}$, and then $\mathcal{F}(\sigma^{min}, \mathcal{D})$ was found.

EXAMPLE 3. Figure 3 shows an example of the Data Distribution approach. The database $\mathcal{D}$ in Figure 1 was divided into two partitions, $\delta_1$ and $\delta_2$. In the first phase, $\mathcal{N}_1$ and $\mathcal{N}_2$ independently apply the basic algorithm on $\delta_1$ and $\delta_2$, respectively. After both nodes finish the first phase, $\mathcal{N}_1$ will know $\mathcal{F}(\sigma^{min}, \delta_1)$ and $\mathcal{MF}(\sigma^{min}, \delta_1)$, while $\mathcal{N}_2$ will know $\mathcal{F}(\sigma^{min}, \delta_2)$ and $\mathcal{MF}(\sigma^{min}, \delta_2)$. In the second phase, they exchange their maximal frequent collections, so that both nodes will know $\bigcup_{i=1}^{2} \mathcal{MF}(\sigma^{min}, \delta_i) = \{1235, 1245\}$. In the third phase, each node performs the top-down enumeration. $\mathcal{N}_1$ process itemsets $\{1245, 124, 145, 245\}$, while $\mathcal{N}_2$ does not process any further itemset (since $\mathcal{MF}(\sigma^{min}, \delta_2) = \bigcup_{i=1}^{2} \mathcal{MF}(\sigma^{min}, \delta_i)$). In the fourth phase, $\mathcal{N}_1$ and $\mathcal{N}_2$ exchange $\pi(\mathcal{X}, \delta_1)$ and $\pi(\mathcal{X}, \delta_2)$ of each itemset $\mathcal{X}$, so that they know all $\mathcal{X}$ with $\sigma(\mathcal{X}, \mathcal{D}) \geq \sigma^{min}$. For instance, $\pi(124, \delta_1) = 1$, $\pi(124, \delta_2) = 3$, and $\sigma(124, \mathcal{D}) = \frac{4}{10} = 0.4$.

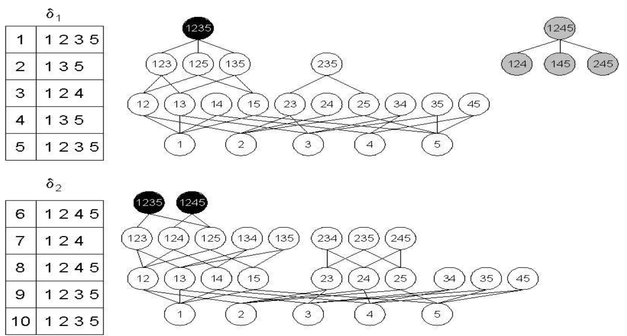


**Figure 3. Data Distribution Example.**

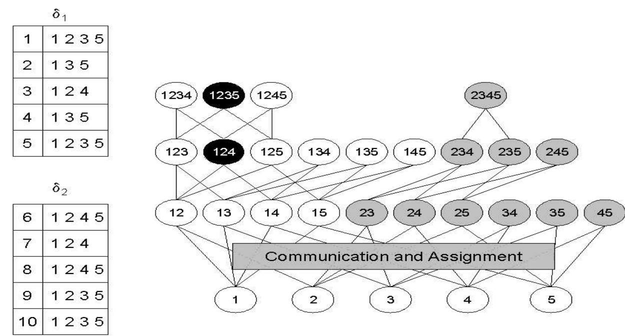


**Figure 4. Candidate Distribution Example.**

**3.1.2. The Candidate Distribution Approach** The basic idea of this approach is to separate the candidates in disjoint sets. Each set is assigned to a node, so that it can asynchronously process its tasks. There are four distinct phases:

PHASE 1. [FIRST ASYNCHRONOUS PHASE] Each node $\mathcal{N}_i$ reads its local partition $\delta_i$, and constructs the tidsets for all items in $\delta_i$.

PHASE 2. [COMMUNICATION AND ASSIGNMENT PHASE] Each node has to exchange the local tidsets with every other node. After each node knows the tidsets of the items in all partitions, we start the assignment of the tasks to the nodes. Note that each item corresponds to a different backtrack tree, which corresponds to a disjoint set of candidate itemsets. The way the backtrack trees are assigned to different nodes depends on the implementation (we will discuss more about this in the next section). After the assignment, there is no dependence among the nodes.

PHASE 3. [SECOND ASYNCHRONOUS PHASE] Each node proceeds the search for frequent itemsets within the backtrack trees assigned to it, using our basic algorithm. Since the dependence among the processors was decoupled in the previous phase, there is no need for further synchronization, and the costly communication of tidsets (in the previous phase) is amortized in this phase.

PHASE 4. [REDUCTION PHASE] Finally, a join-reduction operation is performed and $\mathcal{F}(\sigma^{min}, \mathcal{D})$ was found.

EXAMPLE 4. Figure 4 shows an example of the Candidate Distribution approach. In the first phase, $\mathcal{N}_1$ reads $\delta_1$ and constructs $\mathcal{L}_{\delta_1}(1)$, $\mathcal{L}_{\delta_1}(2)$, $\mathcal{L}_{\delta_1}(3)$, $\mathcal{L}_{\delta_1}(4)$, and $\mathcal{L}_{\delta_1}(5)$. At the same time, $\mathcal{N}_2$ reads $\delta_2$ and constructs $\mathcal{L}_{\delta_2}(1)$, $\mathcal{L}_{\delta_2}(2)$, $\mathcal{L}_{\delta_2}(3)$, $\mathcal{L}_{\delta_2}(4)$, and $\mathcal{L}_{\delta_2}(5)$. In the second phase, both nodes exchange its local tidsets, and then the backtrack tree for item $\{1\}$ is assigned to $\mathcal{N}_1$, while backtrack trees for items $\{2,3,4\}$ are asigned to $\mathcal{N}_2$. In the third phase, each node applies the basic algorithm on the backtrack trees assigned to it (i.e., $\mathcal{N}_1$ has processed the white and bold itemsets, while $\mathcal{N}_2$ has processed the shaded itemsets).

## 3.2. Algorithm Implementation

Two dominant paradigms for using multiple processors have emerged: distributed- and shared-memory. The performance-optimization objectives for distributed-memory implementations are different from those of shared-memory implementations. In the distributed-memory paradigm synchronization is implicit in communication, so the goal becomes communication optimization. In the shared-memory paradigm, synchronization stems from locks and barriers, and the goal is to minimize these points. A third, very popular, paradigm combines the best of these two paradigms. Clusters of SMPs are part of this third paradigm. The physical memory is distributed among the nodes, but each processor within each node has free access to the entire memory of the node. Clusters of SMPs necessitate a hierarchical parallelism implementation, with shared-memory primitives used in a node and message passing used among the nodes. A fourth paradigm has also emerged, which allows a number of clusters to be connected together through a high speed network to work like a single super-computer.

### 3.2.1. The Shared-Memory Implementation

*Cache Locality, False Sharing, and Synchronization:* Our algorithms avoid the use of complex structures, which may have poor locality. In fact, they use simple intersection operations to determine the frequent itemsets. This feature enables the algorithms to have good cache locality. Also, a problem unique to shared-memory systems is false sharing, which occurs when two different shared variables are located in the same cache block, causing the block to be exchanged between the processors even though the processors are accessing different variables. A simple solution would be to place unrelated data that might be accessed simultaneously on separate cache lines. While this simple approach will eliminate false sharing, it will result in unacceptable memory space overhead, and most importantly, a significant loss in cache locality [8]. Another technique for eliminating

false sharing is called *Privatization* [3]. It involves making a private copy of the data that will be used locally, so that operations on that data do not cause false sharing. This technique allows us to keep a local array of frequent itemsets per processor. Each processor can use its local array during the search for frequent itemsets. Privatization eliminates false sharing completely, and there is no need of any kind of synchronization among the processors. We implemented the Candidate Distribution approach using this technique. As explained earlier, the main idea is to assign distinct backtrack trees to distinct processors. Each backtrack tree corresponds to a disjoint set of itemsets, and by using the privatization technique, there is no dependence among the processors. The backtrack trees are assigned to the processors in a *bag of tasks* approach, that is, given a *bag* of backtrack trees to be processed, each processor takes one tree, and as soon as it finishes the search for frequent itemsets on this tree, it takes another tree from the *bag*. When the *bag* is empty, we have found $\mathcal{F}(\sigma^{min}, \mathcal{D})$. We observed that the amount of work associated with a given backtrack tree is generally proportional to the support of the generating item (or root). We sorted the backtrack trees in the bag in descending order of support of their roots, so that bigger tasks will be processed earlier.

**3.2.2. The Distributed-Memory Implementation** We implemented both Data and Candidate Distribution approaches using the distributed-memory paradigm. While data distribution deals with the large size problem, candidate distribution deals with the high dimension problem. The distributed-memory implementation of the Candidate Distribution approach is similar to the shared-memory implementation, that is, different backtrack trees are assigned to different nodes. Next we will describe how the Data Distribution approach was implemented.

*Reducing the Large Size Problem:* One step in the Data Distribution approach deserves special attention: the top-down enumeration. It is very important to implement this step in an efficient way, otherwise one node can replicate work, computing $\pi(\mathcal{X},\delta_l)$ for the same itemset $\mathcal{X}$ more than once. As mentioned before, we can do this by simply intersecting the tidsets of all items that compose $\mathcal{X}$. However, this approach is not efficient because it would need $k-1$ intersections (if the size of $\mathcal{X}$ is $k$). To solve this problem, we store the intermediate tidsets in a hash-table (whose key is the itemset). For example, suppose we must find $\pi(123, \delta_1)$. We first perform the operation $\mathcal{L}_{\delta_1}(1) \cap \mathcal{L}_{\delta_1}(2) = \mathcal{L}_{\delta_1}(12)$, and then $\mathcal{L}_{\delta_1}(12) \cap \mathcal{L}_{\delta_1}(3) = \mathcal{L}_{\delta_1}(123)$, and $\pi(123, \delta_1) = |\mathcal{L}_{\delta_1}(123)|$. After $\mathcal{L}_{\delta_1}(12)$ is processed, it is stored in the hash-table, so that we do not need to process $\mathcal{L}_{\delta_1}(12)$ again.

### 3.2.3. The Hierarchical Implementation

*Using both Shared- and Distributed-Memory:* We implemented a hierarchical version of the Candidate Distribution approach, where different backtrack trees are assigned to different nodes, and inside each node, its backtrack trees are assigned to its processors. This approach greatly reduces the amount of communication performed. We also implemented a hybrid approach, which combines data distribution among the nodes and candidate distribution inside each node. This hybrid approach reduces data skewness, since there are fewer partitions (although bigger) to be processed.

### 3.2.4. The Massively Parallel Implementation

*Balancing Communication and Data Skewness:* A very important issue is how to reduce communication and data skewness. Unfortunately, we cannot reduce both of them at the same time (if communication is reduced, data skewness is increased, and vice-versa). Therefore, we must rely on how to balance these two metrics. This is especially important in massively parallel environments, because if we perform only candidate distribution, then the amount of communication will be too large. Otherwise, if we perform only data distribution, then there will be a large number of partitions, and data skewness will be too high. What is needed is a way to use both Data and Candidate Distribution approaches. We implemented this hybrid approach in such a way that data is distributed among the clusters and candidates are distributed among the nodes within each cluster. This choice reduces the communication among the clusters. Note that Data Distribution approach needs much less communication than Candidate Distribution approach, since only $\bigcup_{i=1}^{n} \mathcal{MF}(\sigma^{min}, \delta_i)$ is transfered among the clusters. Further, the number of partitions will be always the same as the number of clusters involved in the mining process (which is much smaller than the number of processors), reducing data skewness.

## 4. Experimental Evaluation

Our experimental evaluation was carried out on two 8 node PENTIUM processor clusters. In one of the clusters all nodes have two processors. Each node has 1GB of main memory and 120GB of disk space. Writes have a latency of 1.2 $\mu$secs, with transfer bandwidth of 100MB/sec. All nodes inside a cluster are interconnected through a high-speed network, the $Myrinet$. The clusters are interconnected through an optic-fiber. We have implemented the parallel algorithms using the MPI message-passing library (MPICH), and POSIX PTHREADS.

We used a real database for evaluating the performance of our algorithms. The WPORTAL database is generated from the click-stream log of a large Brazilian web portal. We scanned the log and produced a transaction file (i.e., $\mathcal{D}$), where each transaction is a session of access to the site by a client. Each item in the transaction is a web request, but not all web requests were turned into items; to become an

item, the request must have three properties: (1) the request method is GET; (2) the request status is OK; and (3) the file type is HTML. A session starts with a request that satisfies the above properties, and ends when there has been no click from the client for 30 minutes. WPORTAL has 3,183 items comprised in 7,786,137 transactions. In each experiment, WPORTAL was divided into $n$ partitions, where $n$ depends on the number of processors (or clusters) employed.

Our evaluation is based on two basic parameters: number of processors and minimum support. Thus, for each minimum support employed, we performed multiple executions of the algorithms, where each execution employs a different number of processors. Further, we employed four different metrics in our evaluation:

**Execution Time:** It is the total elapsed time spent for mining $\mathcal{D}$. Timings are based on wall clock time.

**Communication:** It is the total amount of bytes transfered among the nodes during the mining operation.

**Parallel Efficiency:** It is given by $\frac{2\times(T-T')}{T}$, where $T$ is the execution time obtained when using $n$ processors, and $T'$ is the execution time obtained when using $2\times n$ processors. For instance, if $T$ = 100 secs and $T'$ = 50 secs, then the parallel efficiency is 1 (100%).

**Data Skewness:** This metric quantifies how even (or uneven) the frequent itemsets are distributed among the partitions. Intuitivelly, if the frequent itemsets are evenly distributed among the partitions, then $\bigcup_{i=1}^{n} \mathcal{MF}(\sigma^{min}, \delta_i)$ (i.e., the upper bound) is close to $\mathcal{MF}(\sigma^{min},\mathcal{D})$. We used the well established notion of entropy [4] to develop a suitable measure for data skewness. For a random variable, the entropy is a measure of the non-uniformity of its probability distribution. Suppose $\delta_1 \cup \delta_2 \cup ... \cup \delta_n = \mathcal{D}$. In this case, the value $p_{\mathcal{X}}(i) = \frac{\sigma(X,\delta_i)}{\sigma(X,\mathcal{D})}$ can be regarded as the probability of occurrence of $\mathcal{X}$ in $\delta_i$. We first define the skewness of an itemset $\mathcal{X}$ as $\mathcal{S}_{\mathcal{X}} = \frac{log(n)-\mathcal{H}_{\mathcal{X}}}{log(n)}$, where $\mathcal{H}_{\mathcal{X}} = \sum_{i=1}^{n} p_{\mathcal{X}}(i) \times log(p_{\mathcal{X}}(i))$. Then the data skewness metric can be defined as a weighted sum of the skewness of all itemsets, that is, $\mathcal{S} = \sum_{X\in F(\sigma^{min},D)} \mathcal{S}_{\mathcal{X}} \times w_{\mathcal{X}}$, where $w_{\mathcal{X}} = \frac{\sigma(\mathcal{X},\mathcal{D})}{\sum_{Y\in F(\sigma^{min},D)} \sigma(Y,\mathcal{D})}$. Please, refer to [4] for further explanations and insights about this metric.

In the first series of experiments we used the 8 node dual processor cluster. In all graphs, *Data Distribution (Distributed)* refers to the distributed-memory implementation of Data Distribution approach ($\mathcal{D}$ is divided into $n$ partitions, where $n$ is the number of processors), *Candidate Distribution (Distributed)* refers to the distributed-memory implementation of Candidate Distribution approach ($n$ partitions), *Data Distribution (Hierarchical)* refers to the hierarchical implementation of the Data Distribution approach ($\frac{n}{2}$ partitions, since there are 2 processors per node), and *Candidate Distribution (Hierarchical)* refers to the hierarchical implementation of the Candidate Distribution approach ($\frac{n}{2}$ partitions). The first experiment we conducted was to empirically verify the advantages of our parallel algorithms in terms of execution times. We varied the number of processors (1 to 16) and the minimum support (the number of frequent itemsets generated varied from 91,874 (for $\sigma^{min} = 0.01$) to 283,186 (for $\sigma^{min} = 0.005$)), and compared the execution times for each parallel algorithm. Figure 5 shows the execution times obtained for different parallel configurations. In all cases the distributed implementations are slightly better when the number of processors is less than 8. If more processors are used, the hierarchical implementations become better, since communication (in the Candidate Distribution approach) and data skewness (in the Data Distribution approach) become too high in the pure distributed implementations. Further, the efficiency increases when we reduce the minimum support, since the asynchronous phase becomes more relevant [10]. The Data Distribution approaches seem to be the best ones. Figure 6 shows the results obtained in a similar experiment, but now the appraised metric is the amount of communication performed by each algorithm. The Candidate Distribution approaches require approximatelly 3-4 orders of magnitude more communication than the Data Distribution approaches. Further, the hierarchical implementations save communication requirements by a factor of 5.

In the next experiment, we investigated how data skewness increases as we vary the number of processors (and consequently, partitions) using the Data Distribution approach (note that the Candidate Distribution approaches do not generate data skewness, since the data is communicated among the processors). As showed in Figure 7, data skewness is higher for smaller minimum supports. Also, as expected, data skewness is higher for the pure distributed implementations, since there are two times more partitions in this case. In the last experiment of this series, we investigated how data skewness affects parallel efficiency. Figure 8 shows that the effect of data skewness are worse for the pure distributed implementations. Although data skewness increases as we reduce the minimum support, parallel efficiency increases for lower minimum support values (since the asynchronous phase becomes more relevant).

The other series of experiments employed the two clusters, but to make "equal" the computational power of both clusters, we used only 4 dual nodes of the first cluster. Our objective is to evaluate our algorithms in terms of execution times and speedup. We varied the number of processors and the minimum support value, and in each parallel configuration employed, each cluster used the same number of processors (i.e., 2 processors mean that cluster 1 and cluster 2 used 1 processor). Figure 9 shows the execution times ob-

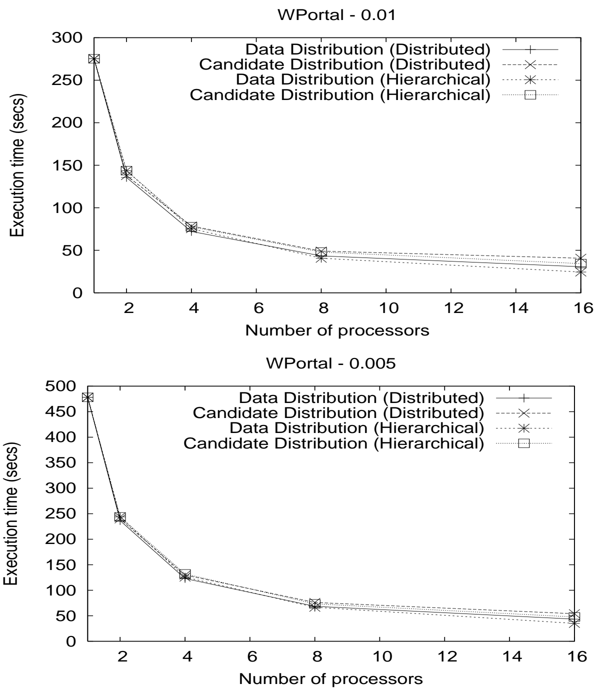


**Figure 5. Total Execution Times as a function of number of processors (SMP cluster).**

tained. For $\sigma^{min} = 0.01$ we observed a speedup number of 11/16, while for $\sigma^{min} = 0.005$ the speedup number reached almost 14/16. These numbers shows that combining both Data and Distribution approaches in the inter-cluster parallel environment can be an efficient approach, since it reduces communication among the clusters, and reduces data skewness, since there are only two logical partitions.

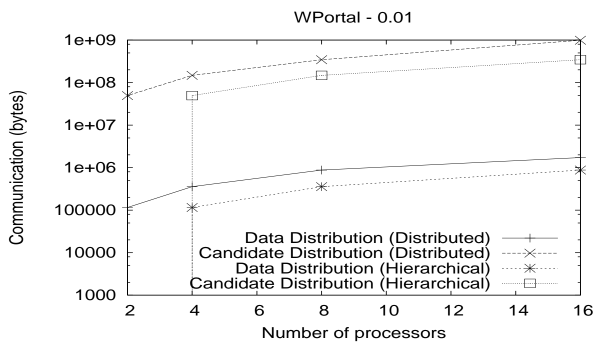


**Figure 6. Communication as a function of number of processors.**

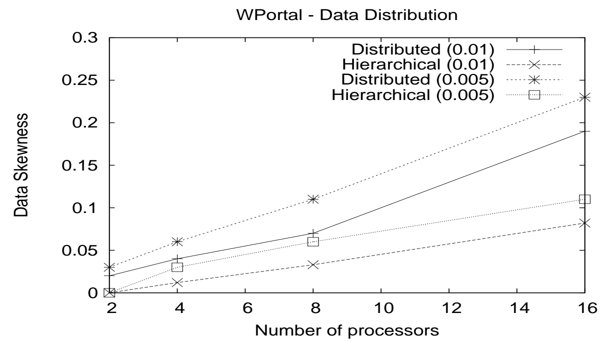


**Figure 7. Data Skewness as a function of number of processors.**

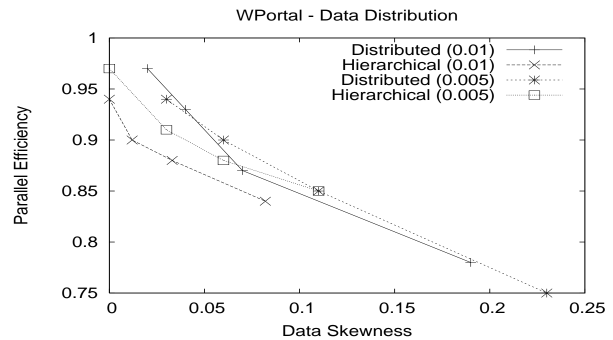


**Figure 8. Parallel Efficiency as a function of Data Skewness.**

## 5. Related Work

Several algorithms for mining frequent itemsets were proposed in the literature [2, 12, 11]. APRIORI [2] was the first efficient algorithm and it forms the core of almost all current algorithms. During the first pass over $\mathcal{D}$ the support of each item is counted. The frequent items are used to generated candidate 2-itemsets. $\mathcal{D}$ is scanned again to obtain the support of all candidate 2-itemsets, and the frequent ones are selected to generate candidate 3-itemsets. This iterative process is repeated for $k$ = 3,4,..., until there are no more frequent $k$-itemsets to be found. Clearly, APRIORI needs $k$ passes over $\mathcal{D}$, incurring in high I/O overhead. In the parallel case, APRIORI based algorithms do a reduction operation at the end of each pass, thus incurring also in high synchronization cost. Three different parallel implementations of the APRIORI algorithm on IBM-SP2, a distributed-memory machine, were presented in [1]. In [10] the authors showed the impact in synchronization

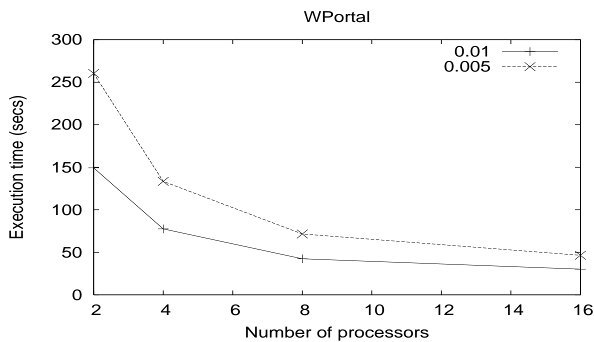


**Figure 9. Total Execution Times as a function of number of processors (2 clusters).**

overhead due to fine-grained parallel approaches. The parallel algorithms NPA, SPA, HPA, and HPA-ELD, proposed in [7] are similar to those in [1]. HPA-ELD is the best among NPA, SPA, and HPA, because it reduces communication by replicating candidates with high support on all processors. ECLAT, an algorithm presented in [12], needs only two passes over $\mathcal{D}$ and decomposes the search-space for frequent itemsets in disjoint sub-spaces. In [13] several ECLAT-based parallel algorithms were presented. Our basic algorithm need only one access to $\mathcal{D}$. In the parallel case, our algorithms need only one communication round. Also, complementar experiments show that our basic algorithm generates a smaller number of candidates than both APRIORI- and ECLAT-based algorithms.

## 6. Conclusions

The huge size of the available databases and their high dimension make data mining applications very computationally demanding, to an extent that high performance parallel computing is fast becoming an essential component of the solution. In fact, data mining applications are poised to become the dominant consumers of supercomputing and high performance systems in the near future. There is a practical necessity to develop effective and efficient parallel algorithms for data mining. In this paper we presented several parallel algorithms for frequent itemset mining, a fundamental data mining task. Key issues such as load balancing, communication reduction, attention to data skewness, improving cache locality and reducing false sharing, were all addressed. Our parallel algorithms need only one access to the disk-resident database and are based on a novel and efficient backtrack algorithm (which was also presented in this paper). Also, our algorithms are the first ones able to deal with both large size and high dimension problems. The algorithms were evaluated on different parallel environments, and showed to be very efficient. We also presented a possible application; mining large web logs to better understand web patterns. We intend to continue our work by distributing the databases in a widely distributed scenario, and developing proper algorithms to deal with the challenges imposed by this scenario.

## References

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

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 20$^{th}$ Int. Conf. Very Large Databases, VLDB*, pages 487–499. Morgan Kaufmann, Dec 1994.

[3] R. Bianchini and T. LeBlanc. Software caching on cache-coherent multiprocessors. In *Proc. Symp. on Parallel and Distributed Processing, SPDP*, pages 521–526. IEEE, May 1992.

[4] D. W.-L. Cheung and Y. Xiao. Effect of data distribution in parallel mining of associations. *Data Mining and Knowledge Discovery*, 3(3):291–314, 1999.

[5] J. Gray and P. Shernoy. Rules of thumb in data engineering. In *Proc. Int. Conf. on Data Engineering, ICDE*, pages 3–12. IEEE, May 2000.

[6] E.-H. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. *Transactions on Knowledge and Data Engineering*, 12(3):728–737, 2000.

[7] M. Joshi, E.-H. Han, G. Karypis, and V. Kumar. Effi cient parallel algorithms for mining associations. *Parallel and Distributed Systems*, 1759:418–429, 2000.

[8] S. Parthasarathy, M. Zaki, M. Ogihara, and W. Li. Parallel data mining for association rules on shared-memory systems. *Knowledge and Information Systems*, 3(1):1–29, 2001.

[9] A. Veloso, W. Meira, M. Bunte, S. Parthasarathy, and M. Zaki. Mining frequent itemsets in evolving databases. In *Proc. Int. Conf. on Data Mining, SDM*, pages 31–41. SIAM, May 2002.

[10] A. Veloso, W. Meira, M. Bunte, S. Parthasarathy, and M. Zaki. Parallel, incremental and interactive frequent itemset mining. In *Proc. Int. Work. on High Performance Data Mining, HPDM*, pages 81–90. SIAM, May 2003.

[11] M. Zaki and S. Parthasarathy. A localized algorithm for parallel association mining. In *Proc. Symp. on Parallel Algorithms and Applications, SPAA*, pages 120–128. ACM, Aug 1997.

[12] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for mining association rules. In *Proc. 2$^{nd}$ Int. Conf. on Knowledge Discovery and Data Mining, SIGKDD*, pages 14–24. ACM, Aug 1997.

[13] M. 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, 1997.