Toward Terabyte Pattern Mining

An Architecture-conscious Solution

Gregory Buehrer
The Ohio State University
buehrer@cse.ohio-state.edu

Srinivasan Parthasarathy
The Ohio State University
srini@cse.ohio-state.edu

Shirish Tatikonda
The Ohio State University
tatikond@cse.ohio-state.edu

Tahsin Kurc
The Ohio State University
kurc@bmi.ohio-state.edu

Joel Saltz
The Ohio State University
saltz@bmi.ohio-state.edu

Abstract

We present a strategy for mining frequent itemsets from terabyte-scale data sets on cluster systems. The algorithm embraces the holistic notion of architecture-conscious data mining, taking into account the capabilities of the processor, the memory hierarchy and the available network interconnects. Optimizations have been designed for lowering communication costs using compressed data structures and a succinct encoding. Optimizations for improving cache, memory and I/O utilization using pruning and tiling techniques, and smart data placement strategies are also employed. We leverage the extended memory space and computational resources of a distributed message-passing cluster to design a scalable solution, where each node can extend its meta structures beyond main memory by leveraging 64-bit architecture support. Our solution strategy is presented in the context of FPGrowth, a well-studied and rather efficient frequent pattern mining algorithm. Results demonstrate that the proposed strategy result in near-linear scaleup on up to 48 nodes.

Categories and Subject Descriptors H.2.8 [*Database Management*]: Database Applications - Data Mining

General Terms Algorithms, Performance

Keywords itemset mining, data mining, parallel, out of core

PPoPP'07 March 14–17, 2007, San Jose, California, USA.

1. Introduction

"Data mining, also popularly referred to as knowledge discovery from data (KDD), is the automated or convenient extraction of patterns representing knowledge implicitly stored or catchable in *large* data sets, data warehouses, the Web, (and) other *massive information repositories or data streams*" [14]. This statement, from a popular textbook, and its variants resound in numerous articles published over the last decade in the field of data mining and knowledge discovery. As a field, one of the emphasis points has been on the development of mining techniques that can scale to truly large data sets (ranging from hundreds of gigabytes to terabytes in size). Applications requiring such techniques abound, ranging from analyzing large transactional data to mining genomic and proteomic data, from analyzing the astronomical data produced by the Sloan Sky Survey to analyzing the data produced from massive simulation runs.

However, mining truly large data is extremely challenging. As implicit evidence, consider the fact that while a large percent of the papers describing data mining techniques often include a statement similar to the one quoted above, a small fraction actually mine large data sets. In many cases, the data set fits in main memory. There are several reasons why mining large data is particularly daunting. First, many data mining algorithms are computationally complex and often scale non-linearly with the size of the data set (even if the data can fit in memory). Second, when the data sets are large, the algorithms become I/O bound. Third, the data sets and the meta data structures employed may exceed disk capacity of a single machine.

A natural cost-effective solution to this problem that several researchers have taken [2, 9, 13, 19, 20, 27] is to parallelize such algorithms on commodity clusters to take advantage of distributed computing power, aggregate memory and disk space, and parallel I/O capabilities. However, to date none of these approaches have been shown to scale

to terabyte-sized data sets. Moreover, as has been recently shown, many of these algorithms are greatly under-utilizing modern hardware[11]. Finally, many of these algorithms rely on multiple database scans and oftentimes transfer subsets of the data around repeatedly.

In this work, we focus on the relatively mature problem domain of frequent itemset mining[1]. Frequent itemset mining is the process of enumerating all subsets of items in a transactional database which occur in at least a minimum number of transactions. It is the precursor phase to association rule mining, first defined by Agrawal and Srikant[3]. Frequent itemset mining also plays an important role in mining correlations [5], causality [24], sequential patterns [4], episodes [17], and emerging patterns [10].

Our solution embraces one of the fastest known sequential algorithms (FPGrowth), and extends it to work in a parallel setting, utilizing all available resources efficiently. In particular, our solution relies on an algorithmic design strategy that is cache-, memory-, disk- and network- conscious, embodying the comprehensive notion of *architecture-conscious data mining*[6, 11]. Our solution is placed in the context of a cache- and I/O-conscious frequent pattern mining algorithm recently developed, which has been shown to be be efficient on large data sets.

A highlight of our parallel solution is that we only scan the data set twice, whereas all existing parallel implementations based on the well-known *Apriori* algorithm [4] require a number of data set scans proportional to the cardinality of the largest frequent itemset. The sequential algorithm targeted in this work makes use of tree structures to efficiently search and prune the space of itemsets. Trees are generally not readily efficient, when local trees have to be exchanged among processors, because of the pointer-based nature of tree structure implementations. We devise a mechanism for efficient serialization and merging of local trees called *strip-marshaling and merging*, to allow each node to make global decisions as to which itemsets are frequent. Our approach minimizes synchronization between nodes, while also reducing the data set size required at each node. Finally, our strategy is able to address situations where the meta data does not fit in core, a key limitation of existing parallel algorithms based on the pattern growth paradigm[9].

Through empirical evaluation, we illustrate the effectiveness of the proposed optimizations. Strip marshaling of the local trees into succinct encoding affords a significant reduction in communication costs, when compared to passing the data set. In addition, local tree pruning reduces the data structure at each node by a factor of up to 14-fold on 48 nodes. Finally, our overall execution times are an order of magnitude faster than existing solutions, such as *CD, DD, IDD* and *HD*.

2. Challenges

In this section, we first present challenges associated with itemset mining of large data sets in a parallel setting. The frequent pattern mining problem was first formulated by Agrawal *et al.* [1] for association rule mining. Briefly, the problem description is as follows: Let $I = \{i_1, i_2, \cdots, i_n\}$ be a set of n items, and let $D = \{T_1, T_2, \cdots, T_m\}$ be a set of m transactions, where each transaction T_i is a subset of I. An itemset $i \subseteq I$ of size k is known as a k-itemset. The $support$ of i is $\sum_{j=1}^{m}(1 : i \subseteq T_j)$, or informally speaking, the number of transactions in D that have i as a subset. The frequent pattern mining problem is to find all $i \in D$ with $support$ greater than a minimum value, $minsupp$.

In most parallel algorithms, the data set D=$D_1 \cup D_2 \cup ... \cup D_n$, $D_i \cap D_j$=$\emptyset$, $i \neq j$, is distributed or partitioned over n machines. Each partition D_i is a set of transactions. An itemset that is globally frequent *must be* locally frequent in at least one D_i. Also, an itemset not frequent in any D_i cannot be globally frequent, and an itemset locally frequent in all D_i must be globally frequent. The goal *of the parallel itemset mining problem* is to mine for set of all frequent itemsets in a data set that is distributed over different machines. Several fundamental challenges must be addressed when mining itemsets on such platforms.

2.1 Computational Complexity

The complexity of itemset mining primarily arises from the exponential (over the set of items) size of the itemset lattice. In a parallel setting, each machine operates on a small local partition to reduce the time spent in computation. However, since the data set is distributed, global decision making (e.g., computing the support of an itemset) becomes a difficult task. Each machine must spend time communicating the partially mined information with other machines. One needs to devise algorithms that will achieve load balance among machines while minimizing inter-machine communication overheads.

2.2 Communication Costs

The time spent in communication can vary significantly depending on the type of information communicated; data, counts, itemsets, or meta-structures. Also, the degree of synchronization required can vary greatly. For example, one of the optimizations to speed up frequent itemset mining algorithms is to eliminate candidates which are infrequent.The method by which candidates are eliminated is called *search space pruning*. Breadth-first algorithms prune infrequent itemsets at each level. Depth-first algorithms eliminate candidates when the data structure is projected. Each projection has an associated context, which is the parent itemset.

When the data is distributed across several machines, candidate elimination and support counting operations require inter-machine communication since a global decision must be reached to determine result sets. There are two fundamen-

tal approaches for reaching a global decision. First, one can *communicate all the needed data, and then compute asynchronously*; An alternate strategy is to *communicate knowledge after every step of computation*. In the former strategy, each machine sends its local portion of the input data set to all the other machines. While this approach minimizes the number of inter-machine communications, it can suffer from high communication overhead when the number of machines and the size of the input data set are large. The communication cost increases with the number of machines due to the broadcast of the large data set. In addition to communication overhead, this approach scales poorly in terms of disk storage space. Each machine requires sufficient disk space to store the aggregated input data set. For extremely large data sets, full data redundancy may not be feasible. Even when it is feasible, the execution cost of exchanging the entire data set may be too high to make it a practical approach.

In the latter strategy, each machine computes local data and a merge operation is performed to obtain global support counts. Such communication is carried out after every level of the mining process. The advantage of this approach is that it scales well with increasing processors, since the global merge operation can be carried out in $O(N)$, where N is the size of the global count array. However, it has the potential to incur a high communication overhead because of the number of communication operations and large message sizes. As the candidate set size increases, this approach might become prohibitively expensive.

Clearly there are trade-offs between these two different approaches. Algorithm designers must compromise between approaches that copy local data globally, and approaches which retrieve information from remote machines as needed. Hybrid approaches may be developed in order to exploit the benefits from both strategies.

2.3 I/O Costs

When mining very large data sets, care should also be taken to reduce I/O overheads. Many data structures used in itemset mining algorithms have sizes proportional to the size of the data set. With very large data sets, these structures can potentially exceed the available main memory, resulting in page faults. As a result, the performance of the algorithm may degrade significantly. When large, out-of-core data structures need to be maintained, the degree of this degradation is in part a function of the temporal and spatial locality of the algorithm. The size of meta-structures can be reduced by maintaining less information. Therefore, algorithms should be redesigned to keep the meta-structures from exceeding the main memory by reorganizing the computation or by redesigning the data structures [11].

2.4 Load Imbalance

Another issue in parallel data mining is to achieve good computational load balance among the machines in the system. Even if the input data set is evenly distributed among machines or replicated on each machine, the computational load of generating candidates may not be evenly distributed. As the data mining process progresses, the number of candidates handled by a machine may be (significantly) different from that of other machines. One approach to achieve load balanced distribution of computation is to look at the distribution of frequent itemsets in a sub-sampled version of the input data set. A disadvantage of this approach is that it introduces overhead because of the sub-sampling of the input data set and mining of the sub-sampled data set. Another approach is to dynamically redistribute computations for candidate generation and support counting among machines when computation load imbalance exceeds a threshold. This method is likely to achieve better load balance, but it introduces overhead associated with redistributing the required data.

No.	Transaction	Sorted Transaction with Frequent Items
1	f, a, c, d, g, i, m, p	a, c, f, m, p
2	a, b, c, f, l, m, o	a, c, f, b, m
3	b, f, h, j, o	f, b
4	b, c, k, s, p	c, b, p
5	a, f, c, e, l, p, m, n	a, c, f, m, p
6	a, k	a

Table 1. A transaction data set with $minsup = 3$

Parallel itemset mining offers trade-offs among computation, communication, synchronization, memory usage, and also the use of domain-specific information. In the next section, we describe our parallelization approach and its associated optimizations.

3. FPGrowth in Serial

FPGrowth proposed by Han *et al* [15] is a state-of-the-art algorithm frequent itemset miner. It draws inspiration from Eclat [26] in various aspects. Both algorithms build meta-structures in two database scans. They both traverse of the search space in depth-first manner, and employ a pattern-growth approach. We briefly describe *FPGrowth*, since it is the algorithm upon which we base our parallelization.

The algorithm summarizes the data set in the form of a prefix tree, called an *FPTree*. Each node of the tree stores an item label and a count, where the count represents the number of transactions which contain all the items in the path from the root node to the current node. By ordering items in a transaction based on their frequency in the data set, a high degree of overlap is established, reducing the size of the tree. *FPGrowth* can prune the search space very efficiently and can handle the problem of skewness by projecting the data set during the mining process.

A prefix tree is constructed as follows. The algorithm first scans the data set to produce a list of frequent 1-items. These items are then sorted in frequency descending order. Following this step, the transactions are sorted based on the order from the second step. Then, infrequent 1-items are pruned away. Finally, for each transaction, the algorithm

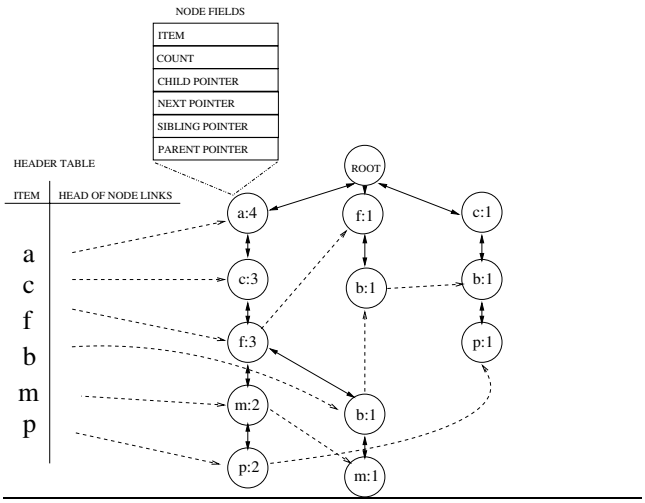

Figure 1. An FP-tree/prefix tree

inserts each of its items into a tree, in sequential order, generating new nodes when a node with the appropriate label is not found, and incrementing the count of existing nodes otherwise.

Table 1 shows a sample data set, and Figure 1 shows the corresponding tree. Each node in the tree consists of an *item*, a *count*, a *nodelink ptr* (which points to the next item in the tree with the same item-id), and *child ptrs* (a list of pointers to all its children). Pointers to the first occurrence of each item in the tree are stored in a header table.

The frequency count for an itemset, say ca, is computed as follows. First, each occurrence of item c in the tree is determined using the node link pointers. Next, for each occurrence of c, the tree is traversed in a bottom up fashion in search of an occurrence of a. The count for itemset ca is then the sum of counts for each node c in the tree that has a as an ancestor.

4. Parallel Optimizations

In this work, we target a distributed-memory parallel architecture, where each node has one or more local disks and data exchange among the nodes is done through message passing. The input data set is evenly partitioned across the nodes in the system. We first present our optimizations, and then in Section 4.5 we discuss how these optimizations are combined in the parallel implementation.

4.1 Minimizing Communication Costs

One of the fundamental challenges in parallel frequent itemset mining is that global knowledge is required to prove an itemset is not frequent. It can be shown that if an itemset is locally frequent or infrequent in every partition, then it is globally frequent or infrequent, respectively. However, in practice, most itemsets lie in between; they are locally frequent in a nonempty proper subset of the partitions and infrequent in the remaining partitions. These itemsets must have their exact support counts from each partition summed for a decision to be made. In Apriori-style algorithms, this is not an additional constraint, because this information is present. Apriori maintains exact counts for each potentially frequent candidate at each level. However, *FPGrowth* discards infrequent itemsets when projecting the data set.

It is too expensive for a node to maintain the count information so that it can be retrieved by another node. This would equate to mining the data set at a support of one. Alternatively, synchronizing at every pass of every projection of each equivalence class is also excessive. Our solution is for each machine to communicate its local *FPTree* to all other machines using a ring broadcast protocol. That is, each node receives an *FPTree* from its left neighbor in a ring organization, merges the received tree with its local *FPTree*, and passes the tree received from its left neighbor to its right neighbor.

Exchanging the *FPTree* structure reduces communication overhead because the tree is typically much smaller than the frequent data set (due to transaction overlap). To further reduce the volume of communication, instead of communicating the original tree, each processor transfers a concise encoding, a process we term *strip marshaling*. The tree is represented as a array of integers. To compute the array (or encoding), the tree is traversed in depth first order, and the label of each tree node is appended to the array in the order it is visited. Then, for each step back up the tree, the negative of the support of that node is also added. We use the negative of the support so that when decoding the array, it can be determined which integers are tree node labels and which integers are support values. Naturally, we again negate the support value upon decoding, so that the original support value is used.

This encoding has two desired effects. First, the encoded tree is significantly smaller than the entire tree, since each tree node can be represented by two integers only. Typically, *FPTree* nodes are 48 bytes, but the encoding requires only 8 bytes, resulting in a 6-fold reduction in size. Second and more importantly, because the encoded representation of the tree is in depth-first order, each node can traverse its local tree and add necessary nodes online. Thus, merging the encoded tree into the existing local tree is efficient, as it only requires one traversal of the each tree. When the tree node exists in the encoding but not in the local tree, we add it to the local tree (with the associated count); otherwise we add the received count (or support) to the existing tree node in the local tree. Because the processing of the encoded tree is efficient, we can effectively overlap communication with processing. Also, transferring these succinct structures dispatches our synchronization requirements. The global knowledge afforded allows each machine to subsequently process all its assigned tasks independently.

For example, Figure 2 illustrates the tree encoding for the tree from our example in Section 2. Note that the labels of

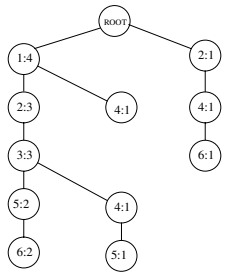

Figure 2. *Strip Marshaling* the *FPTree* provides a succinct representation for communication.

the tree nodes are recoded as integers after the first scan of the input data set.

4.2 Pruning Redundant Data

Although the prefix tree representation (*FPTree*) for the projected data set is typically quite concise, there is not a guaranteed compression ratio with this data structure. In rare cases, the size of the tree can meet or exceed the size of the input data set. A large data set may not fit on the disk of a single machine. To alleviate this concern, we prune the local prefix tree of unnecessary data. Specifically, each node maintains the portion of the *FPTree* required to mine the itemsets assigned to that node.

Recall that to mine an item i in the *FPTree*, the algorithm traverses every path from any occurrence of i in the tree up to the root of the tree. These upward traversals form the projected data set for that item, which is used to construct a new *FPTree*. Thus, to correctly mine i only the nodes from occurrences of i to the root are required.

Let the items assigned to machine M be I. Consider any $i \in I$, and let n_i be a node in the *FPTree* with its label. Suppose a particular path P in the tree of length k is

$$P \quad = \quad (n_1, n_2, ..., n_i, n_i + 1, ..., n_k). \qquad (1)$$

To correctly mine item i, only the portion of P from n_1 to n_i is required for correctness. Therefore, for every path P in the *FPTree*, M may delete all nodes n_j occurring after n_i. This is because the deleted items will never be used in any projection made by M. In practice, M can start at every leaf in the tree, and delete all nodes from n_k to the first occurrence of any item $i \in I$ assigned to M. M simply evaluates this condition on the *strip marshaled* tree as it arrives, removing unnecessary nodes.

As an example, we return to the *FPTree* described in Section 2. Assume we adopt a round-robin assignment function, so machine M is assigned all items i such that

$$i \quad \% \quad |Cluster| \quad = \quad rank(M). \qquad (2)$$

We illustrate the local tree for each of the four machines in Figure 3. A nice property of this scheme is that as we increase the number of machines in the cluster, we increase the amount of pruning available.

4.3 Partitioning the Mining Process

We determine the mapping of tasks to machines before any mining occurs. Researchers[9] have shown that intelligent allocation can in this fashion can lead to efficient load balancing with low process idle times, in particular with *FP-Growth*. We leverage such sampling techniques to assign *frequent-one* items to machines after the first parallel scan of the data set.

4.4 Memory Hierarchy-conscious Mining

Local tree pruning lowers the required main memory requirements. However, it may still be the case that the size of the tree exceeds main memory. In such cases, mining the tree can incur page faults. To remedy this issue, we incorporate several optimizations to improve the performance. Through detailed studies, we discovered that locality improvements in both the tree building process, and the subsequent mining process lead to improved execution times for large data sets [6].

Designers of OS paging mechanisms work under the assumption that programs will exhibit good locality, and store recently accessed data in main memory accordingly. Therefore, it is imperative that we find any existing temporal locality and restructure computation to exploit it. Further details of the following procedures are available elsewhere [6, 11].

4.4.1 Improving the Tree Building Process

The initial step in *FPGrowth* is to construct a prefix tree. For out-of-core data sets, construction of the first tree results in severe performance degradation. If the first tree does not fit in main memory, the algorithm can spend up to 90% of the execution time building it. The reason is that transactions are in the database in random order, which results in random access to the tree nodes during construction, and excessive page faulting. To solve this problem, we incorporate domain knowledge and the frequency information collected in the first scan to intelligently place the frequent transactions into a partition of blocks. Each block is implemented as a separate file on disk. The hashing algorithm guarantees that each transaction in block$_i$ sorts before all transactions in block$_{i+1}$, and the maximum size of a block is no larger than a preset threshold. By blocking the frequent data set, we can build the tree on disk in fixed chunks. A block as well as the portion of the tree being updated by the block will fit in main memory during tree construction, reducing page faults considerably. In short, transactions with the most frequent items are allocated a larger portion of the partition. Further details are available in a prior publication[6].

4.4.2 Improving the Mining Process

Mining the large tree results in significant page faulting because for any two connected nodes, it is unlikely that they

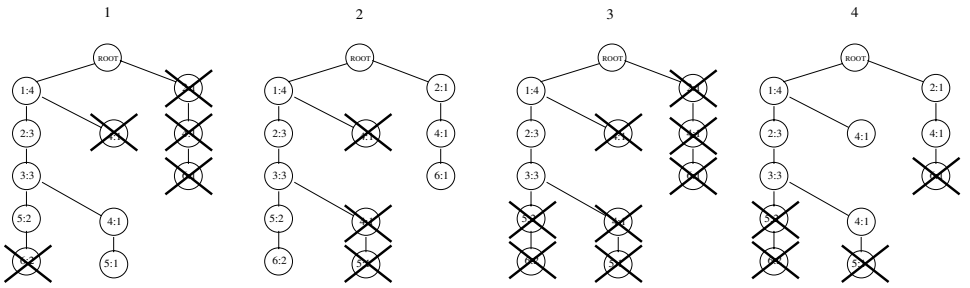

Figure 3. Each node stores a partially overlapping subtree of the global tree.

are near each other in memory. We improve spatial locality by reallocating the tree in virtual memory, such that the new tree allocation is in depth-first order. We $malloc()$ fixed sized blocks of memory, whose sum is equal to the total size of the tree. Next, we traverse the tree in depth-first order, and (in one pass) copy each node to the next location (in sequential order) in the newly allocated blocks of virtual memory. This simple reallocation strategy provides significant improvements, because *FPGrowth* accesses the prefix tree many times in a bottom up fashion, which is primarily a depth-first order of the tree.

Finally, to improve temporal locality while accessing the tree, we tile paths of the tree. Our approach relies on *page blocking*, and is analogous to our tiling techniques for improving temporal locality in in-core pattern mining algorithms. Since each tree is traversed once for each frequent item at that projection, we can walk a small percentage of the tree paths for each item, before continuing to the next set of paths. This improves the probability that the paths will be in cache.

4.5 Putting It All Together: Architecture-conscious Data Mining

We now use the optimizations above to construct our solution. The approach is shown in Algorithm 1, and we label it DFP (Distributed FPGrowth).

The machines are logically structured in a ring. In phase one (lines 1 - 5), each machine scans its local data set to retrieve the counts for each item. Each machine then sends its count array to its right-neighbor in the ring, and receives the counts from its left neighbor. This communication continues *n-1* times until each count array has been witnessed by each machine. The machines then perform a voting procedure to statically assign frequent-one items to particular nodes. The mechanism for voting is designed to minimize load imbalance.

In phase two (lines 6 - 12), each machine builds a prefix tree of its local data set using the global counts derived from phase 1. Then, each machine encodes the tree as an array of integers. These arrays are circulated in a ring manner, as was performed with the count array. Since each machine requires only a subset of the total data to mine its assigned elements, upon receiving the array the local machine incorporates only the contextually pertinent parts of the array into its local tree before passing it to its right-neighbor.

In phase three (lines 13 - 14), each machine independently mines its assigned items. No synchronization between machines is required during the mining process. The results are then aggregated.

Algorithm 1 DFP

Input: Data set D=$D_1 \cup ... \cup D_n$, *Global* Support σ
Output: F = Set of frequent itemsets

1: **for** each node i=1 to n **do**
2: Scan D_i for item frequencies, LF_i
3: **end for**
4: Aggregate LF_i, i=1..n (using a ring)
5: Assign itemsets to nodes
6: **for** each node i=1 to n **do**
7: Scan the D_i to build a *local* Prefix Tree, T_i
8: Encode T_i as an array, S_i
9: Prune T_i of unneeded data
10: **end for**
11: Distribute S_i, i=1..n (using a ring)
12: and build a pruned global tree at each node
13: Locally mine the pruned global tree
14: Aggregate the final results (using a ring)

5. Experiments

We implement our optimizations in C++ and MPI. The test cluster has 64 machines connected via Infiniband, 48 of which are available to us. Each compute machine has two 64-bit AMD Opteron 250 single core processors, 2 RAID-0 250GB SATA hard drives, 8GB of RAM, and runs the Linux Operating System. We are afforded 100GB (per machine) of the available disk space. For this work, we only use one of the two available processors. All synthetic data sets were created with the IBM Quest generator, with 100,000 distinct items, an average pattern length of 8, and 100,000 patterns. Other settings were defaults. The number of transactions varies depending on the experiment, and will be expressed herein. Our main data set is 1.1 terabytes, distributed over 48 machines (24GB each), called 1TB. The real data sets we

use, namely *Kosarak, Mushroom and Chess*, are available in the FIMI repository [12]. Their average transaction lengths are 8, 23 and 37, respectively.

As a comparison, we have implemented *CD, DD*[2] and *IDD*[13]. We should point out that CD never exceeded the available main memory of a compute machine, for all data sets we evaluated (due to the large amount of available main memory per machine). As a result, *HD*[13] is never better than CD.

5.1 Evaluating Storage Reduction

To evaluate the effectiveness of pruning the prefix trees, we record the average local tree size and compare against an algorithm that maintains the entire tree on each machine. We consider the performance of pruning with respect to three parameters, namely the transaction length, the chosen support and the number of machines (see Figure 6).

First, we consider transaction length. Since pruning trims nodes between the leaves and the first assigned node, we suspect that increasing transaction length will degrade the reduction. The number of compute machines for these experiments was 48, and the support was set to be 0.75%. The data set was created with the same parameters as 1TB, except that the transaction length is varied. It can be seen that performance degrades gradually from 16-fold to 12-fold when increasing transaction lengths from 10 to 40 on synthetic data. Degradation in pruning was dampened by the average frequent pattern length, which was not altered. As shown in Figure 6 (middle), the real data sets with longer average transaction lengths generally exhibit less pruning. Recall, the three real data sets have average transaction lengths of 8 (Kosarak), 23 (Mushroom) and 37 (Chess).

We next consider the impact of support on tree pruning. In Figure 6 (middle), we vary the support from 1.8% to 0.5% for Kosarak and 1TB, from 18% to 5% for Mushroom, and from 40% to 5% for Chess. The difference in chosen supports is due to the differences in data set densities. In all cases, we use 48 machines. It can be seen in the middle figure that support only slightly degrades the reduction ratio. This is because although some paths in the tree increase in length (and are thus less prunable), more short paths are introduced, which afford increased prunability. Therefore, as we lower support, the technique continues to prune effectively.

Finally, we evaluate pruning with respect to number of machines used. From Figure 6, it can be seen that the relative amount of reduction increases with an increasing number of machines. For example, at 8 machines we have a 3-fold reduction, but using 48 machines it increases to a 15-fold reduction. We see more improvement from this optimization as the number of machines is increased because each machine has fewer allocated items, thus more tree nodes may be removed from the paths. We found that the variance in local tree size is larger for real data than synthetic data.

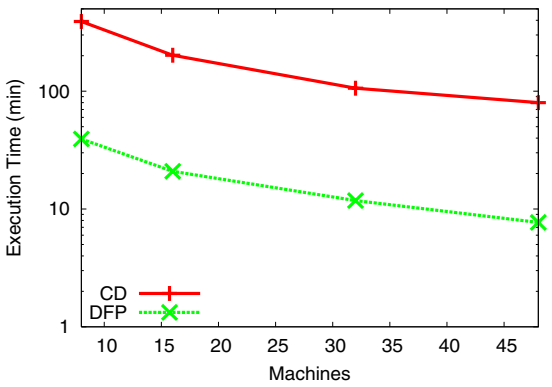

Figure 4. Speedup for 500GB at support = 1.0%.

5.2 Evaluating Communication Reduction

To evaluate the cost of communication, we perform a weak scalability study. Using 8 machines, we mine 1/6th of the data set, and using 48 machines we mine the entire 1TB data set. Each machine contains a 24GB partition. We are interested in measuring the amount of network traffic generated by communicating local trees. Table 2 displays our results, mining at two different supports. The column labeled *Global T'* is the size of the local tree if we maintain full global knowledge at each node. The column labeled *Local T* is the average size of the local tree if we maintain only local data at each node. The column labeled *Local T'* is the average size of the local tree if we maintain partial global knowledge at each node.

Several issues are worth noting. First, the aggregate size of communication for the process (total transferred) is much smaller than the size of the data set. For example, at a support of 0.75% on 48 machines, we transfer about 19 gigabytes. Second, the prefix tree affords a succinct representation of the data set. For each transfer, we save 24GB/2.39GB=10-fold due to the tree structure. Third, tree encoding affords an additional communication reduction. We save an additional 6-fold due to this *strip marshaling*. Also, we make the transfer only once, whereas *IDD* must perform the transfer at each level (at this support, that is 11 levels in all). We do note that in some cases, such as the 8-machine case, it may be possible to improve the excessive traffic of *IDD* by writing some of the data to local storage.

5.3 Evaluating Weak Scalability

To evaluate weak scalability, we employ the same experimental setup. We consider execution time performance as the data set is progressively increased proportional to the number of machines. We mined this data set at two different support thresholds, as shown in Figure 5. The latter support required significantly larger data structures (see Table 2). *DD* suffers greatly from the fact that it transfers the entire data set to each machine at each level. These broadcasts

Machines	Data set(GB)	support%	Global T' (GB)	Local T' (GB)	Local T (GB)	encoding (MB)	total transferred (GB)
8	192	1.00	1.66	0.54	0.32	91.63	0.72
16	384	1.00	3.23	0.57	0.32	96.75	1.51
32	768	1.00	4.87	0.55	0.31	93.33	2.92
48	1152	1.00	6.77	0.58	0.32	98.77	4.63
8	192	0.75	7.33	2.37	1.65	403.63	3.15
16	384	0.75	13.59	2.38	1.63	406.87	6.36
32	768	0.75	21.31	2.39	1.65	408.58	12.77
48	1152	0.75	27.78	2.37	1.65	405.16	18.99

Table 2. Communication costs for mining 1.1 terabytes of data (data set 1TB).

scale poorly with increasing cluster size. For example, at 8 machines there are 8 broadcasts of 24GB. Scaling to 48 machines, *DD* must make 48 independent broadcasts of 24GB, all of which must be sent to 47 other machines. The Infiniband 10Gbps interconnect is not the bottleneck, as each machine is limited by the disk bandwidth of the sender, which is only approximately 90Mbps.

IDD improves upon *DD* slightly, but the inherent challenge persists. *IDD* uses a ring to broadcast its data. However, it makes little effort to minimize the size of the communicated data, and thus scales poorly on large data sets.

CD is a significant improvement over *IDD*. *CD* performs the costly disk scans in parallel. *CD* pays a penalty due to its underlying serial algorithm, which requires an additional scan of the entire data set at each level of the search space. However, *CD* shows excellent scalability, since it parallelizes the largest portion of the execution time. The algorithm proposed in this paper, *DFP*, performs most efficiently. It leverages the available memory space by building a data structure which eliminates the need to repeatedly scan the disk. Also, it effectively circulates sufficient global information to allow machines to mine tasks independently, but does not suffer the penalties that are incurred by *DD* and *IDD*.

Some small overhead ($<5\%$) is incurred when the number of machines becomes large because disk bandwidth heterogeneity exists, and all algorithms presented scan the data set from disk at least twice.

5.4 Evaluating Strong Scalability

To evaluate strong scalability (or speedup), we partitioned 500GB of synthetic data onto 8, 16, 32 and 48 machines. Thus the amount of data in all trials is 500GB. We then ran the algorithms at various supports. The results are available in Figure 4. Speedups are in relation to the time to mine the data set on eight machines, since we could not fit 500GB on a smaller number of machines. We do not include *DD* or *IDD* in the figure because their performance was extremely poor, as they do not improve the time to transfer the data between machines. For example, on 8 machines *DD* requires 140 minutes just to pass the data to each node. This number does not improve with an increasing number of machines because although the data on each machine is less, more machines must transfer their data to more locations. Therefore, *DD* and *IDD* do not scale well on large data sets.

CD scales admirably, from 391 to 80 minutes when increasing the cluster size from 8 to 48 machines. *DFP* scales even better, from 39.3 to 7.5 minutes on the same sized clusters. Also, *DFP* is about 10-fold faster. We believe this gap will widen when mining dense data sets, because *CD* does not parallelize the candidate generation phase.

6. Related Work

In this section we consider algorithms which mine for frequent itemsets which have been designed for shared-nothing or distributed memory architectures (clusters). For a survey of relevant work in the context of shared memory architectures the interested reader is referred elsewhere [20, 28].

Agrawal and Shafer were the first to address the challenge of frequent itemset mining on shared-nothing architectures, proposing Count Distribution (*CD*) and Data Distribution *DD* [2]. In a later work, Han, Karypis and Kumar [13] proposed Intelligent Data Distribution (*IDD*) and Hybrid Distribution (*HD*), which build upon *CD* and *DD*. Decision Miner [22] was proposed by Schuster and Wolff. It reduces the communication costs of *CD*, by pruning candidates which cannot be frequent. Both *DD* and *CD* are based on *Apriori*. *Apriori* algorithms traverse the itemset lattice in a breadth-first manner. At level k, candidate itemsets of size k are generated by using frequent itemsets of size $k-1$. These candidates are then validated against the database (usually by a full scan) to obtain k-size frequent itemsets. Apriori-based algorithms speed up the computation by using the *anti-monotone* property and by keeping some auxiliary state information. The *anti-monotone* property states that if a size k-itemset is not frequent, then any size $(k+1)$-itemset containing it cannot be frequent.

6.1 CD

Count Distribution (*CD*) lowers disk costs by parallelizing the data set scans. In *CD*, each node scans its local data set to obtain frequencies of candidates. The frequency information is then shared amongst the processors to generate a global frequency array. Once the global frequencies are obtained, each processor generates a new set of candidates for the next level. Note that the set of candidates generated by each processor is same, which is the full set of frequent candidates. The algorithm then repeats, as each node again scans the data set to obtain frequencies for the new candidate list. *CD* reduces the communication between processors at the cost

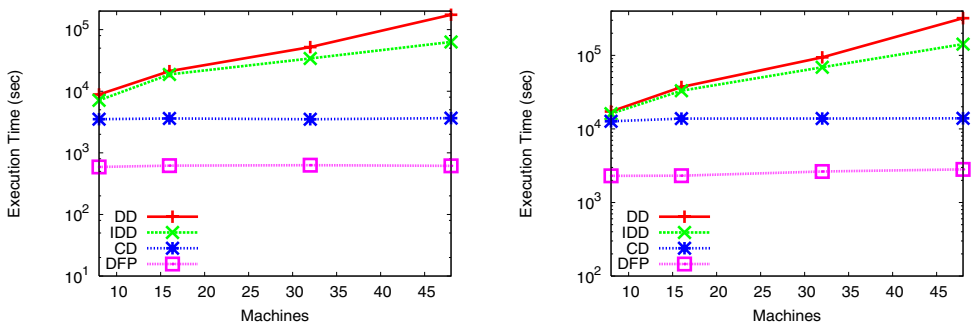

Figure 5. Weak scalability for 1TB at support = 1.0 and 0.75%.

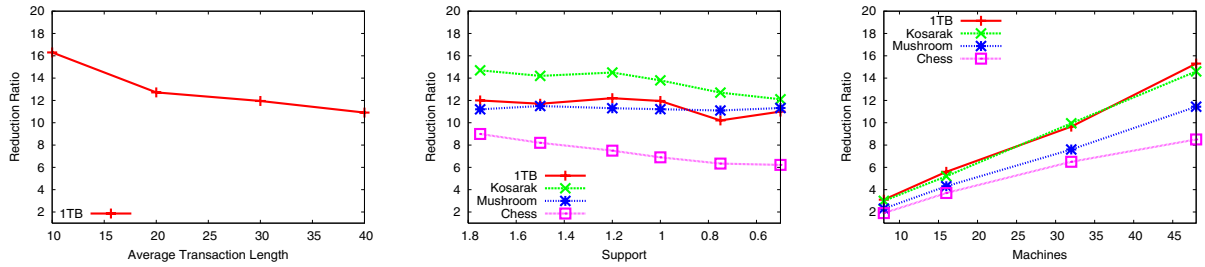

Figure 6. Reduction afforded by tree pruning as a function of transaction length, minimum support, and cluster size.

of redundant computations (in the form of candidate generation) in parallel. *CD* affords excellent scale-up with respect to the number of transactions, because the data set is counted in parallel. However, it scales poorly as the number of candidates increases, since the full list is generated by every node. Candidates tend to increase with decreasing support, as well as with increasing unique items in the data set.

6.2 DD

Data Distribution distributes the data set to each node. At the start of each level, each node takes a turn broadcasting its local data set to every other node. Each machine generates a mutually disjoint partition of candidates. At each level, frequent itemsets are communicated so that each processor can asynchronously generate its share of candidates for the next level. *DD* overcomes the bottleneck of serial candidate generation by partitioning generation amongst the processors. However, it incurs very high communication overhead when broadcasting the entire data set, which is especially costly in case of large data sets. In their study, the authors find the costs of communication outweigh the improvements to parallel candidate generation. We note that in both *CD* and *DD*, processors need to be synchronized at the end of each step, which can potentially cause processors to idle if the data is skewed.

6.3 IDD and HD

Intelligent Data Distribution (IDD) reduces the communication overhead by distributing the data using a ring all-to-all broadcast algorithm. In the original DD, each node broadcasts its data set, which results in unnecessary communication. Although an improvement, *IDD* still suffers from the high overhead of transferring the entire data set at each level.

The authors noted that a weakness in *CD* is that the global candidate list may exceed the size of available main memory for a single node. To address this problem while minimizing communication and computation overheads, Hybrid Distribution (HD) combines the advantages of both the *CD* and *IDD* algorithms by dynamically grouping processors and partitioning the candidate set accordingly to maintain good load balance. Each group is large enough to hold the candidate list in main memory, and executes the *IDD* algorithm. Then between group communications are performed as in *CD*.

6.4 FDM, FPM and DecisionMiner

Cheung *et al.* developed *FDM* [7] as an improvement to *CD*. The main contribution of *FDM* is that it lowers the number of candidates considered for counting. Since the number of candidates is reduced, the cost of communication is lowered as well, because counts for each candidate are broadcast by each node in the cluster. *FDM* accomplishes this reduction by eliminating candidates which are known to be locally infrequent at that site.

The technique was modified to use less polling for a shared-memory version called *FPM* [8]. In both *FPM* and *FDM*, the number of full data set scans is proportional to the length of the longest frequent pattern. *DecisionMiner* [22]

was proposed by Schuster and Wolff. It reduces the communication costs of *CD* by pruning candidates which cannot be frequent. The work is similar to *FDM*, but also provides theoretical bounds for the amount of communication reduction. As does *FDM*, *Decision Miner* makes the same number of full data set scans as *Apriori*.

6.5 Sampling

Schuster, Wolff and Trock [23] parallelized Toivonen's sampling algorithm [25] for a distributed setting, combining it with a common partitioning scheme. This scheme [21] makes the observation that all globally frequent itemsets are locally frequent in at least one partition. The algorithm shows excellent scale-up. In addition, the experimentation is one of the few existing works to mine data larger than a few gigabytes (they mine 16 GB). However, as discussed prior, there is no bound on the number of full data set scans, which can be large when the support is low. Also, the size of the sample at low support thresholds is rather large. Since the algorithm cannot accommodate out-of-core mining, performance degrades on large data at low supports.

6.6 Zig Zag

Otey and Parthasarathy have explored distributed itemset mining on dynamic data sets [18]. By using a backtracking strategy called *ZigZag*, they compute the set of maximal global frequent itemsets.

6.7 Eclat and FPGrowth

Unlike the above approaches, which traverse the search space in a breadth first manner, *Eclat* [26] is an algorithm that traverses the search space in a depth first manner. The idea underlying *Eclat*'s design is to enable the algorithm to chunk out the search space into independent components, and then proceed to evaluate candidates using simple join or intersection operators on transaction lists associated with relevant itemsets. The advantages of this approach, when compared with the Apriori-style approaches, are improved locality, a limit on the number of database scans (two) and a natural structure for parallelization (after the first two database scans – each component can be processed asynchronously independent of the others). However, on some data sets the algorithm suffers from accessing and storing redundant information, and an inability to prune candidates as stringently as the *Apriori*-style algorithms.

FPGrowth [15] addresses the limitations of Eclat through the use of a smart data structure. By employing a projected data set, it is able to eliminate the limitations of *Eclat*. However, while delivering excellent sequential performance, a limitation to parallelizing this algorithm is its reliance on a dynamic, pointer-based data structure and the fact that the data structure can potentially be out-of-core for very large data sets. Javed and Khokhar [16] parallelized FPGrowth for distributed machines. They use a comparable tree pruning approach. Although the communication strategy appears to be polling, they do reduce local tree sizes by transferring only data required at each node. They do not address communication mechanisms and their costs. Results are presented for a small data set (apparently 1 MB) and for up to eight processors on a shared-memory machine.

Zaane, El-Hajj and Lu [29] developed a shared-memory parallelization of FPGrowth. They augment the *FPTree* structure to afford parallel counting. The results show scalability to 64 processors (52-fold) when not factoring in disk I/O. The research shows execution times for a 3GB data set.

Cong *et al* [9] have proposed a sampling based framework for parallel itemset and sequence mining using FP-Growth. They first selectively sample the transactions by discarding a fraction of the most frequent items from each transaction. Based on the mining time on the reduced data set, the computation is divided into equal tasks. In practice, they found that sample mining times were quite representative of actual mining times when mining equivalence classes (*frequent-one items*). Using these timing results, tasks could be assigned to machines statically, while affording excellent load balancing. A drawback of their approach, particularly when mining very large dense data sets is that it assumes that the prefix tree is in-core, and that the data is present on a single central node, which might not be true in many real world scenarios.

The work presented in this paper extends this theme addressing the limitations of previous research.

7. Conclusion

In this paper, we have presented a parallelization of a fast frequent itemset mining algorithm to enable efficient mining of very large, out-of-core data sets. Our implementation employs an "architecture-conscious" design strategy to efficiently utilize memory, processing, storage, and networking resources while minimizing I/O and communication overheads. The salient features include i) a serialization and merging strategy for computing global trees from local trees, ii) efficient tree pruning for better use of available memory space, iii) mechanisms to efficiently handle out-of-core data sets and data structures, and iv) the fact that the input data set is scanned only twice. Our experimental evaluation using a state-of-the-art commodity cluster and large data illustrates a speedup of an order of magnitude when compared to an efficient parallel algorithm, *CD*. The pruning strategy is able to reduce the size of the data by up to 16 times on 48 nodes, and the amount of reduction improves as the number of nodes is increased. Also, the serialization strategy results in a six-fold reduction when exchanging local tree data between nodes.

Acknowledgments

This work is primarily supported by NSF grant #NGS-CNS-0406386, as well as by #CAREER-IIS-0347662 and #RI-CNS-0403342.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 1993.

[2] R. Agrawal and J. Shafer. Parallel mining of association rules. *IEEE Transactions on Knowledge and Data Engineering*, 1996.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1994.

[4] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 1995.

[5] S. Brin, R. Motwani, and C. Silverstein. Beyond market basket: Generalizing association rules to correlations. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 1997.

[6] G. Buehrer, S. Parthasarathy, and A. Ghoting. Out-of-core frequent pattern mining on a commodity pc. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 86–95, New York, NY, USA, 2006. ACM Press.

[7] D. Cheung, J. Han, V. Ng, A. Fu, and Y. Fu. A fast distributed algorithm for mining association rules. In *Proceedings of the International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 31–42, 1996.

[8] D. Cheung, K. Hu, and S. Xia. Asynchronous parallel algorithm for mining association rules on shared-memory multiprocessors. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 279–288, 1998.

[9] S. Cong, J. Han, J. Hoeflinger, and D. Padua. A sampling-based framework for parallel data mining. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 255–265, New York, NY, USA, 2005. ACM Press.

[10] G. Dong and J. Li. Efficient mining of emerging patterns: Discovering trends and differences. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 1999.

[11] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y. Chen, and P. Dubey. Cache-conscious frequent pattern mining on a modern processor. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 577–588, 2005.

[12] B. Goethals and M. Zaki. Advances in frequent itemset mining implementations. In *Proceedings of the ICDM workshop on frequent itemset mining implementations*, 2003.

[13] E.-H. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 277–288, New York, NY, USA, 1997. ACM Press.

[14] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2000.

[15] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2000.

[16] A. Javed and A. Khokhar. Frequent pattern mining on message passing multiprocessor systems. *Distributed and Parallel Databases*, 16:1–14, 2004.

[17] H. Mannila, H. Toivonen, and A. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1997.

[18] M. Otey, C. Wang, S. Parthasarathy, A. Veloso, and W. Meira. Mining frequent itemsets in distributed and dynamic databases. In *Proceedings of the International Conference on Data Mining (ICDM)*, 2003.

[19] S. Parthasarathy and S. Dwarkadas. Shared state for distributed interactive data mining applications. *Journal of Parallel and Distributed Databases*, 2002.

[20] S. Parthasarathy, M. Zaki, M. Ogihara, and W. Li. Parallel data mining for association rules on shared-memory systems. *Knowledge and Information Systems Journal*, 2001.

[21] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1995.

[22] A. Schuster and R. Wolff. Communication-efficient distributed mining of association rules. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 473–484, New York, NY, USA, 2001. ACM Press.

[23] A. Schuster, R. Wolff, and D. Trock. A high-performance distributed algorithm for mining association rules. In *Proceedings of the International Conference on Data Mining (ICDM)*, 2003.

[24] C. Silverstein, S. Brin, R. Motwani, and J. Ullman. Scalable techniques for mining causal structures. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1998.

[25] H. Toivonen. Sampling large databases for association rules. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1996.

[26] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery discovery of association rules. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 1995.

[27] 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*, 1997.

[28] M. J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4):14–25, 1999.

[29] O. R. Zaane, M. El-Hajj, and P. Lu. Fast parallel association rule mining without candidacy generation.