Memory-conscious Frequent Pattern Mining on Modern and Emerging Processors

Implications for Chip Multiprocessor Architectures

Gregory Buehrer Srinivasan Parthasarathy
Amol Ghoting
The Ohio State University
buehrer,srini,ghoting@cse.ohio-state.edu

Yen-Kuang Chen Anthony Nguyen
Daehyun Kim Pradeep Dubey
Intel Corporation
yen-kuang.chen,anthony.d.nguyen
daehyun.kim,pradeep.dubey@intel.com

Abstract

Frequent pattern mining is a fundamental data mining process which has practical applications ranging from market basket data analysis to web link analysis. The challenge in pattern mining is to find all possible groups of items or values which occur in at least σ objects of a database, where σ is user specified parameter. In this work, we show that the state-of-the-art frequent pattern mining algorithms are inefficient when executing on a shared memory multiprocessor system, due primarily to poor utilization of the memory hierarchy. To improve the efficiency of such algorithms, we propose novel techniques designed to afford effective parallelization. Specifically, we present memory performance improvements, task partitioning strategies, and task queuing models designed to maximize the scalability of pattern mining on SMP systems. Empirically, we show that the proposed strategies afford significantly improved scalability and performance. We also discuss implications of this work in light of recent trends in micro-architecture design, particularly chip multiprocessors (CMPs).

1. Introduction

Over the past decade, advances in data collection and storage technologies have resulted in large and dynamically growing data sets at many organizations. Many companies already have data warehouses in the tera-byte range (e.g. FedEx, UPS, Walmart). Similarly, scientific data has reached gigantic proportions (e.g. Genomic data banks). While database technology has provided us with the basic tools for accessing and manipulating such data stores, the issue of how to make end-users understand this data has become a pressing problem. The field of data mining, spurred by advances in data collections and storage technologies, concerns itself with the discovery of knowledge hidden in these large data sets. Today, data mining applications constitute a rapidly growing segment of the commercial and scientific computing domains.

Frequent pattern mining [1] is an immensely popular data mining approach which aims to discover groups of values that co-occur frequently in a data set. There are several instantiations of the frequent pattern mining problem. Examples of these include frequent itemset mining [1], frequent sequence mining [3], frequent subgraph[1] mining [18], and frequent tree mining [7]. While the nature of the patterns being discovered by these instantiations vary, they share a common algorithmic structure. For instance, in frequent itemset mining, we aim to discover groups of items that co-occur frequently in a transactional data set. Similarly, in frequent sub-graph mining we aim to discovery sub-graphs that co-occur frequently in graph data sets.

One challenge in frequent pattern mining is that the search space grows exponentially with the number of items in the data set, forcing runtimes to be quite long. Frequent pattern mining is made even more difficult when mining graphs due to the subgraph isomorphism challenge. A simple chemical data set of 300 molecules can require many hours to mine when the support is set sufficiently low. Parallel architectures can improve these execution times when supplied with a scalable parallel program.

Based on the common algorithmic structure shared by the various frequent pattern mining instantiations, in this paper we identify the following problems in the state-of-the-art. First, frequent pattern mining algorithms typically exhibit poor temporal locality. In addition, spatial locality is poor as well. Most depth-first pattern miners employ pointer-based structures, such as prefix trees. These data meta structures are constructed such that parent and child nodes seldom are on the same cache line. To illustrate, the itemset mining algorithm FPGrowth has been shown to be the fastest algorithm in this domain, yet it has an L2 hit rate of just 43%[11]. Also, in many cases the working sets scale with data set size. This results in poor cache utilization and increased traffic to the main memory sub-system when processing large data sets, limiting scalability on SMP systems. Second, extant parallelization strategies for frequent pattern mining result in significant load imbalance. This is especially true of algorithms that handle complex data types such as graphs. The net result is poor system utilization.

To alleviate the aforementioned bottlenecks, we present various computation re-structuring techniques to sustain high performance with increasing problem size. Specifically, we make the following contributions. First, we present strategies to improve temporal locality in parallel frequent pattern mining algorithms. The proposed optimizations allow for improved scalability on SMP systems due to reduced contention at the memory sub-system. Second, we present a dynamic task partitioning model to handle the load imbalance associated with frequent pattern mining algorithms that handle complex data sets. To realize effective dynamic task parti-

[1] We will use the terms *subgraph* and *substructure* interchangeably.

tioning, we evaluate the use of various task queuing models and show that a distributed queuing model affords the best balance of data locality, concurrency, and locking costs. We empirically evaluate the effectiveness of the proposed techniques on state-of-the-art frequent itemset and frequent sub-graph mining algorithms. While most of our experiments are conducted in the context of SMP systems, we will discuss how this work will influence data mining algorithm design on CMP systems.

The rest of this paper is organized as follows. We will present a brief background in frequent pattern mining together with challenges in Section 2. In Section 3, we will present the proposed optimizations followed by an experimental evaluation in Section 4. A discussion on how we believe this work will influence data mining algorithm design on emerging systems will be presented in Section 5. Related work will be presented in Section 6, and finally, conclusions will be presented in Section 7.

2. Parallel Pattern Mining Challenges

Briefly, the frequent pattern mining problem can be described is as follows: Let $O = \{o_1, o_2, \cdots, o_n\}$ be a set of n objects, and let $D = \{d_1, d_2, \cdots, d_m\}$ be a set of m database entries, where each entry d_i is a subset of O, together with a set of relationships defined amongst the objects. For example, in frequent itemset mining a database entry is a transaction. A pattern is a subset of O, and possibly a set of relationships amongst the objects (edges in graph mining). The $support$ of a pattern p is $\sum_{j=1}^{m}(1 : i \subseteq d_j)$, or informally speaking, the number of entries in D that contain p. The frequent pattern mining problem is to find all patterns in D that have $support$ greater than a minimum support value, $minsupp$. At times we refer to a *frequent one item*, defined to be an instance of the smallest length pattern p found to be frequent in D. There are usually many unique frequent one items for a given problem instance.

Depth-first frequent pattern mining algorithms share a common algorithmic structure, as depicted in Figure 1. These algorithms are recursive in nature. At each step in the recursion, first we find all frequent objects in the database (D). Next, for each of these frequent objects, we append the object to the parent pattern, and construct a projected database $D' \subseteq D$, D' is the subset of D that includes the new frequent object. Finally, this projected database, along with the new pattern, is passed down the recursion. This procedure is carried out for each frequent object in D, at each step in the recursion.

The most popular approach to parallelize a frequent pattern mining algorithm is to partition the work along the *for loop* (Figure 1) at the first level in the recursion. Therefore, in parallel frequent pattern mining, a task involves finding all frequent patterns that start with a certain frequent object[2]. All existing approaches statically or dynamically partition these coarse-grained tasks across the processors of an SMP [33, 15].

To achieve a high level of scalability in a parallel algorithm, designers must overcome several fundamental obstacles. These are load imbalance, efficient utilization of the memory hierarchy, and redundant computation. We outline these issues in the context of pattern mining below.

2.1 Avoiding Load Imbalance

A fundamental challenge when designing parallel algorithms is to maintain a balanced load on the system, such that each node is working at full capacity for the duration of the overall execution. This challenge is exacerbated in pattern mining because the time to mine a task is not known a priori. It is generally accepted to be the greatest challenge when parallelizing any frequent pattern mining algorithm [8].

[2] In order to define the starting object, we usually impose an ordering on the objects in a pattern.

```
Input: A database $D$, minimum support $min_s$
Output: Set of all frequent patterns
Initially: prefix=$\emptyset$

Procedure Find-$\sigma$-Patterns ($D$, prefix, $min_s$)
(1)For each frequent object $o$ in $D$
(2)    Output $o \cup$ prefix as frequent
(3)    Find $D' \subseteq D$ which contains $o \cup$ prefix
(4)    Find-$\sigma$-Patterns($D'$,$o \cup$ prefix, $min_s$)
```

Figure 1. Structure of a typical frequent pattern mining algorithm.

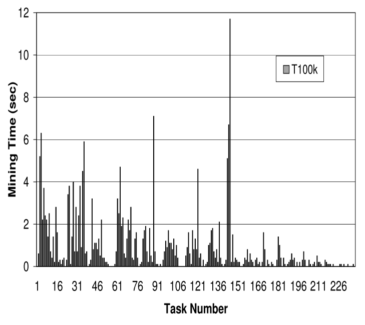
2.1.1 Task Partitioning

The basis for effective load balancing is a task partitioning mechanism possessing sufficient granularity so as to allow each node to continue to perform useful work until the mining process is complete. In itemset mining, for large databases, frequent one items generally suffices. However, for more complex patterns, this is not the case. In substructure mining, a single frequent one item may contain 50% or more of the total execution cost. This is because task length is largely dependent on the associativity in the dataset. In molecular data sets, for example, the edge C-C (representing a carbon-carbon bond) is much more frequent than C-P (a carbon-phosphorus bond), and it will be involved in a majority of the frequent patterns. The histograms in Figure 2 illustrate the degree of imbalance for both synthetic and real world graph data sets. Although both workloads in the Figure show a degree of imbalance, it is clear that the real data set is far less parallelizable, as only a few tasks require almost 100% of the mining time. The synthetic data set affords many more tasks with significant mining times, thus improving its potential for proper load balancing. As our target is real world data, extant task partitioning does not suffice.

2.1.2 Task Allocation

The challenge in task allocation is to develop an allocation strategy that works in conjunction with a partitioning algorithm to eliminate processor idle time. Extant partitioning and allocation strategies for general pattern mining[21] result in significant idle time, which lowers scalability. Broadly speaking, two allocation models exist, static task allocation and dynamic task allocation. Under static allocation, tasks are assigned to nodes *a priori*. Because the total number of tasks in pattern mining is unknown, one must create an assignment function which maps a task to a node. For example, a common allocation scheme is to assign a task to the node which is modulo the task (given an appropriate hashing function); Such static techniques invariably suffer a high performance penalty due to insufficient system load balance.

Several dynamic allocation strategies can be chosen. In global queuing, all processors share a common queue. It affords high sharing, but with high queuing costs. Hierarchical queuing lowers queuing costs because each node has a dedicated queue without a lock. However, since processors can only access queues along their path in the hierarchy, load balance suffers. Distributed queuing attempts to glean the best of both strategies. Each node has a queue with a lock. Nodes enqueue into their own queue, but may steal from other queues if their queue is empty. It should be the case that nodes without work explicitly seek work from other queues, so as to avoid idle nodes. This active stealing minimizes idle time, at the cost of an increase in implementation complexity.



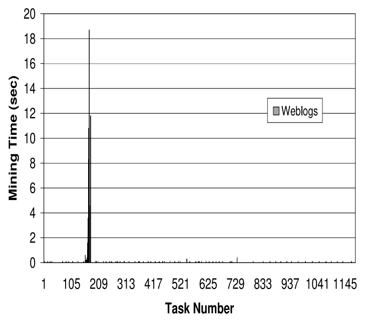


Figure 2. Task time histograms for level-3 partitioning with synthetic data (left) and real data (right).

2.2 Avoiding Poor Memory System Utilization

Frequent pattern mining algorithms exhibit poor spatial and temporal locality[26]. For example, although FPGrowth has been shown to be the fastest itemset mining algorithm for a variety of data sets [13], it has an average L2 hit rate of merely 43% [11]. This is in part due to its traversal of prefix trees, which are pointer-based. These accesses lack temporal locality due to minimal reuse of the projected data set. In addition, these trees grow proportionally with the data set size, thus working sets do not fit in cache. However, because the algorithm accesses the tree in a predictable fashion, locality improvements can increase hit rates. The problem of poor locality is exacerbated in graph mining, because the access pattern of the meta structures used is not predictable.

An additional constraint on shared memory systems is that main memory access comes at a premium. While shared-nothing clusters have the benefit of independent memory systems for each processing node, SMPs processors must vie for the memory bus, and share available main memory. An excessive number of main memory accesses can lead to contention, limiting overall scalability. Programs that exhibit poor data locality typically exhibit high bus utilization because most cache line fetches are not reused. Consequently, we must improve memory system performance to limit contention and maximize scalability on SMP systems.

A final practical concern is that *malloc()* system calls are typically serialized. Although researchers [4] have devised methods to lessen the constraint, parallel allocation of multithreaded programs is still an open problem. Challenges such as fragmentation, false sharing, and blowup hinder these allocators [3].

A final concern with the memory hierarchy is that multiple tasks may share data structures in memory. Until all tasks with a reference to an object are completed, the object must reside in main memory. These shared data structures greatly hinder performance because they consume excessive main memory.

One such structure is an *embedding list*. An embedding is list of mappings between a potentially frequent object and its locations in the database. For itemset mining, embedding lists are quite simple, merely transaction ID lists. However, for graph mining, embedding lists are more expensive because one must store a mapping for each edge in the subgraph. To complicate matters, each subgraph may appear more than once in a database graph. Full embedding lists have been shown to provide a performance improvement [22] in serial graph mining implementations because they allow for fast discovery of child graph candidates. However, the dependence on the memory structure is costly in a parallel setting, because all child graphs of the same parent must be mined before the parent's state can be *free'd*. This effectively creates a synchronization constraint, and stresses the memory subsystem. In addition, full mappings consume excessive memory, limiting the size of the problem which can be solved. An efficient algorithm must explore and optimize the trade off between no embeddings and full embeddings.

2.3 Minimizing Extraneous Work

Given a balanced load and efficient memory system performance, the last major obstacle to an excellent parallel pattern mining algorithm is extraneous work. This extraneous work may come in the form of a) redundant computation, b) additional work due to parallelizing the serial algorithm, and c) queuing costs. We consider redundant work to be computation performed once in a serial algorithm but more than once in a parallel algorithm.

Let W be the work performed by a serial algorithm. In system with n nodes, a parallel implementation can at best do the same work, plus queuing costs q.

(1) $\Sigma_1^n w_i = W + q$

This system would then be performing no extraneous work, save queuing costs. Note that this assumes we have no caching effects.

3. Algorithmic Improvements

In the following section we detail our solutions to the challenges outlined in Section 2. As a base, we start with the state-of-the-art in serial pattern mining; FPGrowth for mining frequent itemsets and gSpan for mining frequent sub-structures. We would like to point out that, Gaston [22], another serial frequent sub-structure mining algorithm, could be considered here. However, we find empirically[4] that its heavy dependence on the use of embedded lists leads to two important problems. First, it places a lot of dependency constraints which limits the available parallelism. Second, it trades off computation for memory, resulting in a very large memory footprint. The second issue has severe implications for bandwidth constrained systems such as CMP systems.

3.1 Improving System Load Balance

3.1.1 Task Granularity

Our solution to handle the task granularity issue is to allow the size of a task to change depending on the state of the system. We term this *dynamic task partitioning*. In dynamic task partitioning, each node makes a decision at runtime for each child; the child may be

[3] http://parasol.tamu.edu/ rwerger/Courses/689/spring2002/day-3-ParMemAlloc/

[4] These results have not been presented due to space constraints.

mined by the creating node at that time, or enqueued as a new task. This decision is based on the current load balance in the system. We do not require a specific mechanism to make this decision, because in part the decision is based on the queuing mechanism used. This will be explained in more detail in section 3.1.2.

For comparison, we implement several static partitioning mechanisms, which we term *level-wise partitioning*. The goal of level-wise partitioning is to create sufficient tasks to allow the work to be balanced between the nodes. Level-wise partitioning deterministically enqueues all children of the frequent one items to a given depth of recursion. For example, in the context of sub-structure mining, when employing level-one partitioning, each frequent one edge is enqueued. These tasks are then mined to completion by the dequeuing node. In level-two partitioning, a task is made for each child of a frequent one edge, and then each task is enqueued. Level-two tasks are then mined to completion by the dequeuing node. Level-n partitioning implies level-(n-1) partitioning. For example, level-5 partitioning will enqueue all tasks 5 levels deep in the search space. Figure 3 compares dynamic partitioning with level-wise partitioning. The level-wise partitioning is set to perform level-2 partitioning (Figure 3(left)). Independent tasks are shaded gray[5]. In Figure 3(right), a task whose size was allowed to grow dynamically has been circled. As we can see from the figure on the right, the parent task $A-B$ had two candidates, namely $A-B-C$ and $A-B-E$. Task $A-B-E$ was enqueued, but $A-B-C$ was mined by the miner of $A-B$. This task then had an additional task $A-B-E$, with two children, both of which were kept and mined by the miner of $A-B$. Thus task $A-B$ dynamically grew based on system input, whereas in the left figure it remained one recursive call.

3.1.2 Task Allocation

In this section we examine our options for a queuing model to accommodate dynamic task allocation. Several queuing models are available; we describe them below. Each queue can be either First-In-First-Out (FIFO) or Last-In-First-Out (LIFO).

A global (or central) queuing model is a model in which each node adds and removes tasks from the same queue. A single mutex is required for all queuing and dequeuing. Contention is an obvious concern, because operations on the queue are serialized. One benefit is that there is maximum task sharing amongst nodes. Any task enqueued into the system is readily available to any idle node. Another benefit is ease of implementation. If a node finishes a task, and there are no other tasks in the global queue, it sleeps. When a node enqueues a task, it awakens all sleeping nodes. Termination occurs when a node finds no tasks in the queue and all other nodes are sleeping.

Hierarchical queuing is designed to allow task sharing between nodes. Each node has its own dedicated queue, which does not have a mutex. Nodes enqueue and dequeue from their own queue. If this queue is full, a node will enqueue into a shared queue. Conceptually, this queue is a level above the dedicated queues. This shared queue can either be above a subgroup of nodes, or over all nodes (i.e. easily extensible to more than two levels). If it is over a subset, then only that subset adds and removes from it. When a node's dedicated queue is empty, it attempts to dequeue from the shared queue. This shared queue has a mutex, and occurs some contention. If the shared queue is for a subset of nodes, then there will be another shared queue above it, which is for multiple node groups. This hierarchical structure ends with a queue with unlimited capacity at the top, which is shared by all queues. A node sleeps when its queue and all queues above it are empty. If a node spills a task from a local queue to a global queue, it awakens the other nodes in the group. Termination occurs when a node finds no other tasks, and all other nodes are sleeping. The hierarchical model allows for task sharing, while affording fast enqueue and dequeue operations in most situations because the local queue has no lock.

[5] As an optimization, each node actually keeps 1 child task, and enqueues the others.

Input: A database D, minimum support min_s
Output: All σ-frequent patterns.
Method: Call Mine().
Initially: $Q = (\sigma$-1 items)

```
Procedure Mine()
(1) while(Q!=empty)
(2)     Frequent Pattern p = Q.dequeue()
(3)     D' = Subset of $D$ containing p
(4)     Find-$\sigma$-Patterns(s,$D'$)

Procedure Find-$\sigma$-Patterns(p,$D$)
(1)if (minimumCode($p$)) //if s is a canonical label
(2)     Add $p$ to Results
(3)     Extension-Patterns C =
(4)     C = Find All single Item subpatterns for $p$ in $D$
(5)     for each Child $c_i$ in C
(6)         if ($c_i$ is frequent)
(7)             Frequent Pattern p' = p + $c_i$
(8)             $D_i'$' = Subset of $D$ containing p'
(9)         if (timeToMine)
(10)                Find-$\sigma$-Patterns(p',$D_i'$)
(11)            else
(12)                Q.enqueue(p',$D_i'$)
```

Figure 4. Pseudo code for dynamic partitioning in frequent pattern mining.

In the distributed queuing model, there are exactly as many queues as nodes. Each queue is unlimited in capacity, has a mutex, and is assigned to a particular node. The default behavior for a node is to enqueue and dequeue using its own queue. Although this incurs a locking cost, there is generally no contention. If a node's queue is empty, it searches other queues in a round robin fashion, looking for work. If all queues are empty, it sleeps. When a node enqueues a task, it wakes all sleeping nodes. Before a node sleeps, it checks whether other nodes are sleeping. If all other nodes are sleeping, the algorithm terminates. The distributed model affords more task sharing than the hierarchical model, at the cost of increased locking. Also, as the size of the local queue decreases in the hierarchical model, it more closely resembles the distributed model.

The basic algorithm is detailed in Figure 4. In line 1 the pattern p is verified to be minimal. This step is required to avoid extraneous work when mining complex patterns having isomorphism, such as general graphs. In line 4 all the objects which contain the pattern p are mined for new, extended patterns. In lines 5-11, all candidate patterns which occur in at least σ database entries are recursively mined. Line 8 checks a boolean condition, which is based on the partitioning scheme and granularity used, to decide whether to mine the child directly or enqueue the task. The insight is that each

The value of the boolean *timeToMine* can be determined using several different schemes. With a global queue, it is natural to check the size of the queue. If it is above a minimum value, then the processor which created the task mines the task without queuing; otherwise it enqueues the child as a new task. With distributed queues, an upper limit on the number of tasks queued in the system is used as a threshold. Hierarchical models set a minimum size for any queue in its hierarchy. A more involved decision could be made based on the rate of decay of the size of a queue, a direction for future work.

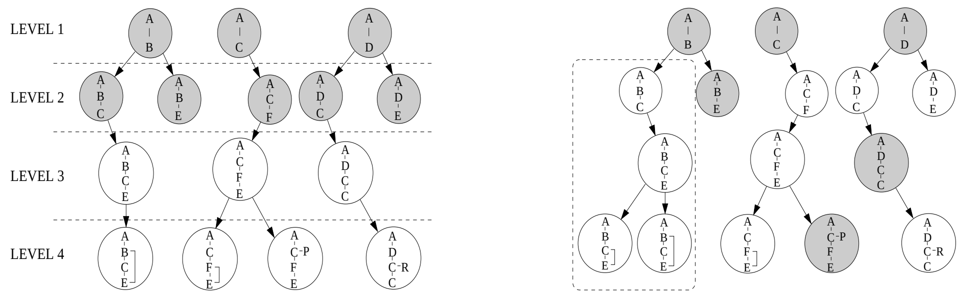


Figure 3. Level-wise Partitioning (left) vs Dynamic Partitioning (right) for graph mining.

3.2 Improvements to Memory System Performance

As discussed previously, frequent pattern mining algorithms exhibit poor temporal locality. This is primarily due to the fact that although the projected data set is re-used when mining for each frequent object, due to the depth-first search space ordering, the re-use is not close in time. Consequently, the projected data set needs to be repeatedly fetched into cache every time it needs to be accessed. There is also a potential for re-using the projected data between iterations of the for loop shown in Figure 1. However, as we process the projected data set for each frequent object, the ordering of the frequent objects being arbitrary, we do not effectively re-use the projected data set. This results in poor temporal locality and poor cache utilization.

In order to improve temporal locality in frequent pattern mining algorithms, one can restructure computation so as to re-use contents of the projected data set once it is fetched into cache. The restructuring is depicted in Figure 5. Essentially, we break down the projected data set into fixed size tiles, and each frequent object is processed on a tile-by-tile basis. This significantly improves temporal locality, improves cache utilization, and minimizes contention on the main memory subsystem.

This methodology has been used to improve temporal locality in frequent itemset mining by breaking down the prefix tree, which in the data structure for the projected data set, into tiles [11]. This is accomplished by re-ordering the tree in depth first order and using an address range in this re-ordered tree to specify a tile.

In graph mining, we improve temporal locality by tiling accesses to the projected graph database. Since multiple candidates search the same data set graphs, and these graphs may be large, we can reduce cache miss rates by searching a data set graph for multiple candidates before loading the next graph into cache.

For example, let g be a frequent subgraph. Let H be the data set graphs containing g. We call H g's embedding list. Let C be a set of candidates for g. Recall that a candidate $c \in C$ is an edge we can add to g to create a new frequent subgraph. This edge may add an additional node as well, or merely create a cycle in g. It is intuitive that each graph $g + c$ must have an embedding list which is a (potentially proper) subset of H. It is also intuitive that we must search each $h \in H$ for each $c \in C$. The data set graphs are typically much larger than the frequent subgraphs. Depth-first graph miners such as Gaston and gSpan follow the structure *for each c in C; for each h in H;*. We reverse this to *for each tile of H; for each c in C;*, where the set of all tiles is a partition of H. By tiling H, we can control the working set. The result is improved temporal locality for accesses to each data set graph h, thus reducing miss rates and improving overall execution time. *We believe that such designs are especially important for CMP systems, which are known to provide limited memory bandwidth per processing core.*

```
Input: A database $D$, minimum support $min_s$
Output: Set of all frequent patterns
Initially: prefix=$\emptyset$

Find-Frequent-Patterns ($D$, prefix, $min_s$)
(1)For each tile $t$ of $D$
(2)    For each frequent pattern $p$ in $D$
(3)        Find $D' \subseteq t$ that has $p \cup$ prefix
(4)        $D'_p = D'_p \cup D'$
(5)For each frequent object $p$ in $D$
(6)    Output $p \cup$ prefix as frequent
(7)    Find-Frequent-Patterns($D'_p$,$p \cup$ prefix, $min_s$)
```

Figure 5. Restructuring frequent pattern mining algorithms to improve locality

We believe that if the work is balanced in the system, having the creator of a task mine the child will always be more efficient than enqueing the task, due to the benefits of affinity [19]. We incorporate this notion into our partitioning algorithm to further improve temporal locality by dynamically growing tasks to include their candidates. As mentioned, graph $g + c$ is always located in a subset of the graphs of its parent g. If the same processing node mines which mines g also mines $g + c$, a percentage of the data set graphs containing g will already be in the working set for that processor. Thus we aggressively attempt to schedule candidates on the same processing node as the parent.

3.3 Improvements to Reducing Extraneous Work

For itemset mining, redundant work is not difficult to avoid. However, general substructure mining is complicated by subgraph isomorphism. To verify a graph g has not been previously mined by any other node, extent parallel strategies must search through a global list of mined items. This is quite expensive and requires regular global aggregation operations. Several labeling and normalization systems have been proposed [22, 31, 20]. We leverage DFScodes [31] because a processor can determine if the graph is redundant without global knowledge of the graphs mined by other nodes (as seen on Line 1 of Figure 4).

4. Experimental Results

We implement our algorithms in C++ using POSIX threads. All experiments allocate one thread per processor, which we term a node.

On a single node, our two pattern mining algorithms compare favorably to the state-of-the-art. We implemented our own gSpan code, since only the binary is accessible. Serial execution times for our gSpan implementation are comparable, we typically take about 20% less time than the authors' binary to process the same data set. Unfortunately, we cannot compare to gSpan on data sets with

Data set	Graphs	Node Lbls	Edge Lbls
Weblogs	31181	9061	1
PTE1	340	66	4
T100k	100000	1000	50

Table 1. Data sets used for the graph mining experiments.

more than 256 node labels, as that was the maximum allowable for the binary provided by the authors of gSpan. The FP-Growth implementation we use is based on recent work described in [11].

We employ several machines to evaluate our work; a) an SGI Altix 3000 with 64 GB of main memory and 32 1.3 Gigahertz Intel Itanium 2 processors, b) an SGI Altix 350 with 32GB of main memory and 16 1.4GHz Intel Itanium 2 processors, and c) a Pentium 4 with 512 MB of main memory. All three machines run Linux.

4.1 Data Sets

For itemset mining, we make use of the Webdocs data set[6]. It is the largest in the FIMI repository, with 1.7 million transaction, 5.3 million unique items, and 1.5GB in size. For graph mining, we employ the data sets shown in Table 1. PTE1 is a data set of molecules classified as carcinogens by the Predictive-Toxicology Evaluation project[7]. HIV1 is a data set of active (CA) molecules which have been screened for anti-HIV1 activity by the National Cancer Institute[8]. Weblogs is a data set of web sessions, generated from web server request and response logs[28]. T100k is a synthetic data set made from the PAFI toolkit[9].

4.2 Load Balance

4.2.1 Dynamic Partitioning

To evaluate our task partitioning strategies, we employ the Weblogs and PTE1 data sets. Both data sets exhibit a high differential for frequent one edge mining times, and are difficult to properly load balance. We use distributed queues for these experiments. The results are presented in Figure 6.

Level-wise partitioning of the search space does not provide an even distribution of the tasks. Even when partitioning to 10 levels into the recursion tree, scalability is hindered. The differences in scalability between the static partitioning models is primarily due to load balancing. Full partitioning enqueues each child task, regardless of the state of the system. As illustrated in Figure 6, *Dynamic Partitioning outperforms all static partitioning models evaluated.* Dynamic partitioning provides 27-fold speedup on 32 processors, whereas levelwise partitioning, even to the 10th level of recursion, can only provide 15-fold speedup.

4.2.2 Dynamic Task Allocation

To evaluate our allocation models, we executed them on several data sets for general substructure mining, using dynamic partitioning. As noted earlier, all three models use dynamic allocation. We present results on the Weblogs data set (Figure 8). Distributed queuing clearly outperforms both global queuing and hierarchical queuing. Contention on the global queue is not a major limiting factor in scalability at 8 nodes, but there is a clear gap between distributed queuing and global queuing. Distributed queuing affords a speedup of 7.85, while global queuing affords a speedup of 7.3. This gap in scalability is expected to widen because contention on the lock for the global queue will increase with increasing nodes. Hierarchical queues perform poorly because task sharing is far too low. As seen in Figure 2, the Weblogs data set has only a few frequent one edges with a significant number of child jobs. These jobs are not propagated to distant nodes in the hierarchical queue structure, particularly those on the opposite side of the hierarchical tree.

[6] http://fimi.cs.helsinki.fi/data/webdocs.pdf

[7] http://web.comlab.ox.ac.uk/oucl/research/

[8] http://dtp.nci.nih.gov/docs/aids/aids_data.html

[9] http://www-users.cs.umn.edu/ karypis/pafi/

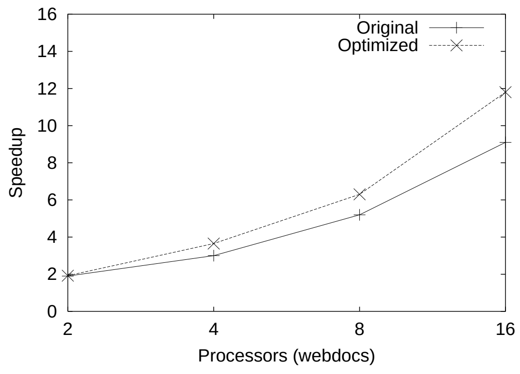


Figure 7. Parallel speedups for itemset mining using on Altix 350 for Webdocs.

FIFO vs LIFO had no statistically significant impact. This is because once a node enqueues a task, it is unlikely to have high temporal locality when it is dequeued, regardless of where it is located in the queue (top or bottom). Some data sets with low frequent one tasks benefit slightly with a FIFO queue because the children of the large tasks are more likely to be large as well, and the sooner they are distributed, the better the load balancing.

4.3 Memory System Performance

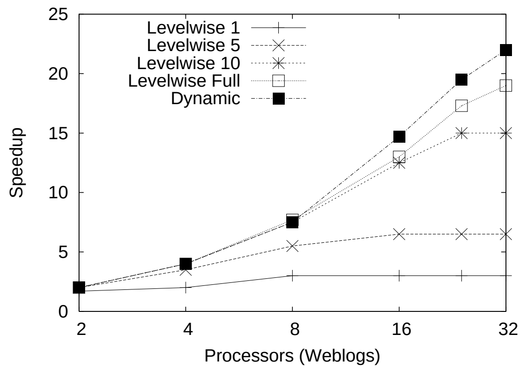
The performance difference between full partitioning and dynamic partitioning is primarily due to the poor cache performance exhibited in full partitioning[10]. We perform a working set study to compare the temporal locality of the two strategies in the context of graph mining. We use Cachegrind [11] on a single processor machine (Pentium 4 2GHz with 1GB RAM). Because Cachegrind currently does not profile multiple threads, we simulate 32 threads by allowing a single thread to remove from any point within 32 locations from the tail of the queue with equal probability. As seen from the results in Figure 9, *dynamic partitioning reduces the miss rates by 50%.* This is due to the improved temporal locality. In dynamic partitioning, there is a much higher probability that the processor which mines a parent task will also mine its child tasks. Child patterns are extensions of parent patterns, thus the database objects which embed a child must be a subset of the database objects which embed its parent. As such, the database objects which embed the child task's pattern are much likelier to be in cache.

In addition, we evaluate our temporal improvements in the context of overall scalability. To measure the scalability of the parallel versions of *FP-Growth* with and without the locality optimizations, we use an SGI Altix 350 system. This SMP has 32GB of main memory and 16 1.4GHz Intel Itanium 2 processors. We report speedups on the Webdocs data set. Note that the speedups reported on our improved implementation are relative to the uniprocessor improved implementation. Similarly, the speedups reported on the parallel un-optimized implementation are relative to the uniprocessor un-optimized implementation.

Figure 7 shows the speedups obtained on 2, 4, 8, and 16 processors for the Webdocs data set. An evaluation on other data sets

[10] This was corroborated through simulation (excluded due to space constraints), where we could eliminate queuing costs

[11] http://valgrind.org/info/tools.html



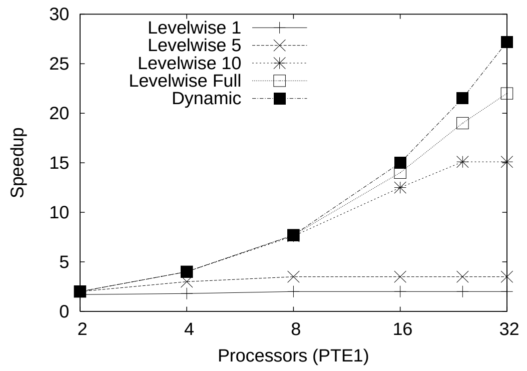
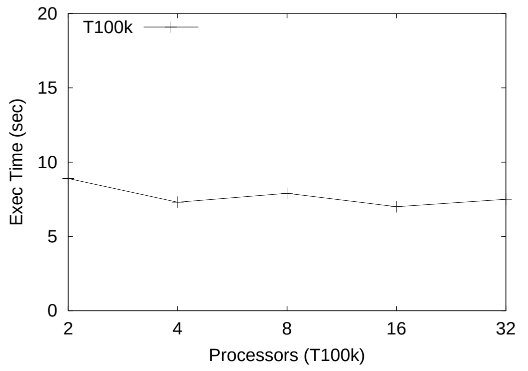


Figure 6. Partition model Scalabilities on Weblogs (left), and PTE1 (right).



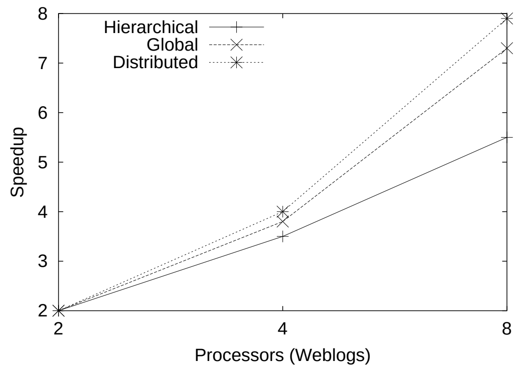


Figure 8. Weak scalability results for T100k (left), and queuing model scalabilities (right).

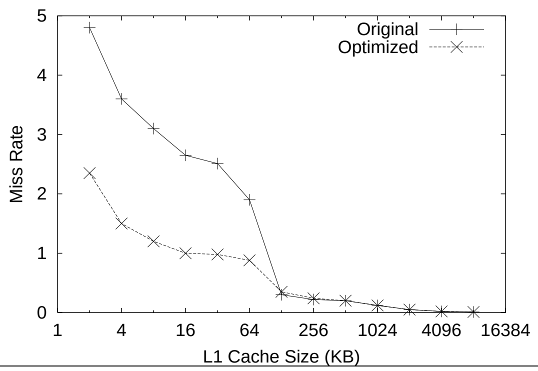


Figure 9. Working sets for full-partitioning vs dynamic partitioning.

can be found elsewhere [12]. We can see that the optimizations afford superior scalability (12x) when compared with those provided by the unoptimized parallel implementation (9x). We attribute this to the improved memory system utilization for the optimized implementation. In the case of the unoptimized implementation, an excessive number of cache misses results in frequent accesses to main memory, potentially resulting in contention. This experiment indicates that such optimized designs are key to achieving good scalability on present-day SMP architectures.

Finally, we evaluate overall memory consumption. Figure 10 provides memory consumption as a function of the number of processors used in the system for graph mining on the PTE1 data set. As a comparison, we provide the memory consumption for parallel[12] MoFa [21], kindly provided by the authors. The illustrated low memory consumption is directly related to the minimal embeddings used. As seen in the Figure, our optimized algorithm (which is based on gSpan [31]) uses a constant amount of memory (5MB), while MoFa's memory consumption increases with the number of threads of executions (300MB on 8 processors). For embeddings, we merely store pointers to nodes in the data set where construction for the current pattern should initiate. As a comparison to serial mining algorithms, this is slightly more state than unoptimized gSpan [31] but much less state than the full embeddings used by Gaston EL [22]. We note that their implementation is in Java. However, it is clear from the documentation of the algorithm that in MoFa, additional threads require additional memory to improve execution time, regardless of the programming language.

4.4 Eliminating Extraneous Work

Our parallel algorithms do not perform work which does not exist in their serial counterparts, save queuing costs; we do not require additional operations to maintain correctness in the parallel algorithm, such as aggregation operations. This has been shown elsewhere [31, 17]. We believe that queuing costs are not a significant impedance because with our distributed queuing model, the common case is to enqueue and dequeue from the local queue.

[12] More information regarding the MoFa can be found at the conclusion of Section 6.

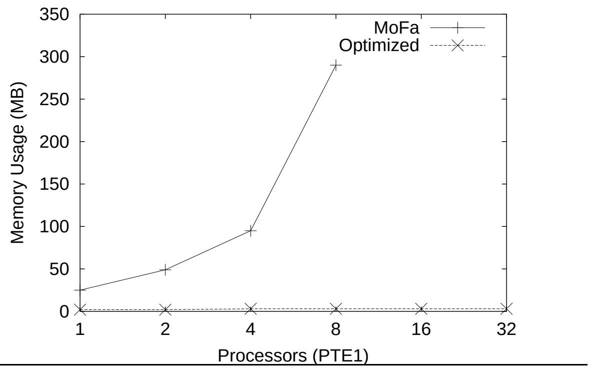


Figure 10. Memory consumption evaluation for graph mining codes on the PTE1 data set.

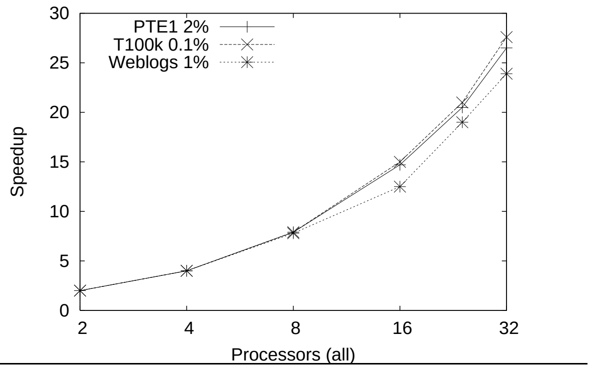


Figure 11. Strong scalability of the optimized graph mining code on four data sets.

4.5 Scalability

We provided scalability for itemset mining in section 4.3. For graph mining, we measure both weak and strong scalability. To measure strong scalability, we used the full data set and increased the number of nodes. On 32 nodes, the scalability ranges from 27.4 to 22.5 over the four data sets, as seen in Figure 11. We attribute the overall excellent scalability to efficient use of the memory system, and a lack of processor idle time afforded by the improved load balancing. Weblogs has the lowest scalability at 22.5. This can be attributed to its increased memory traffic. The frequent graphs in the Weblogs data set are larger than those in the other data sets, which requires the algorithm to traverse a larger percentage of each of the database graphs, deeper into the recursion. Scalability generally increases as support is reduced, because there are more tasks, and those tasks can be mined by the creating node.

We also measure weak scalability of our algorithms. We present weak scalability for graph mining here. We partitioned the synthetic data set T100k into 32 equal chunks, and executed the algorithm with a number of chunks proportional to the number of nodes. Ideally, we would scale work precisely with increasing nodes. Graph mining runtimes are highly dependent on the associativity and distribution of the data, so we tuned support in each experiment to return a graph result set proportional to the number of nodes. We use the synthetic data set T100k for this experiment because we believe its partitions more closely resembles the properties of the full data set. The results are presented in Figure 8. The algorithm clearly accommodates the increase in data set size without a degradation in performance.

5. Discussion and Future Work

Throughout this work, our focus has been on removing the bottlenecks of scalability. we found that poor performance of the memory subsystem greatly reduced overall scalability. Temporal locality was low due to streaming accesses to large data objects. We altered the algorithms to reduce working set sizes through tiling, and ultimately lowered cache misses. In addition, we used succinct data structures to avoid excessive main memory consumption, and we reduced synchronization constraints of shared memory objects to minimize idle time. Also, heap allocation was a challenge. Most operations in graph mining use data containers whose size is not known a priori, so we use heap space. For example, when searching for a graph in the database, it is not known in how many graphs it will be embedded. Our solution was to give each node its own allocation of heap space for recursive calls.

5.1 Implications for CMP Architectures

Chip Multiprocessing (CMP) designs (also known as multicore architectures) incorporate more than one processing element on the same die. They can execute multiple threads concurrently because each processing element has its own context and functional units.

CMP systems have significant differences when compared to existing platforms. To begin with, *inter-process communication costs can be much lower than previous parallel architectures*. CMP systems can keep locks on chip, either in specially designed hardware, or in a shared L2 cache. A direct consequence of lower locking costs is an improved task queuing system. Therefore, *algorithm designers should consider a much finer granularity* when targeting these systems. In this work we have designed a partitioning mechanism which accommodates allocation of work at each level of the mining recursion, which is a very fine grain parallelism. Future work could improve on this for CMP systems by allowing multiple cores to search different graphs for the same candidate. For example, suppose frequent subgraph g has embedding list H. To mine for candidates of g, we must search each graph $h \subseteq H$ for g. We could schedule individual cores to tile sets (or blocks) of H, such that the same cores search the same subsets of H as the algorithm progresses deeper into the search space.

Several important issues with CMP systems arise when discussing the memory hierarchy. We believe that *off chip bandwidth will be limited* in CMP systems, much more so than with SMP designs. While clusters have bandwidth which scales with increasing processing elements, CMP systems do not. CMPs are even more limited than SMPs in this regard, because each node in an SMP has independent access to the memory subsystem. CMP systems are limited by the size of the chip because there are practical limits to the number of pins which can be placed on a single chip. Although not significant with two cores, as systems scale to 32 and 64 cores, off chip bandwidth will be a significant concern. Our path tiling optimization, where the size of each tile can be controlled, lends itself to such an approach. Also, as witnessed in Figure 9, we have chosen a design with low offchip bandwidth requirements. We accomplish a low memory footprint by maintaining very little state for embedding lists. We keep only pointers to seed nodes where occurrences must be grown.

CMP systems will have much smaller caches than their SMP counterparts. This is predicated by the real estate constraints of the chip, since for a fixed chip size each additional core will use silicon previously dedicated to cache. As such, CMP systems require significant redesign to maintain fixed-sized working sets. This allows the algorithm to scale with both the problem size and the number of cores. We accomplish fixed working set sizes by tiling accesses in the main loop of our pattern mining algorithms; in paths of a prefix tree for itemset mining, and graphs in the projected data set for graph mining. Recall in Section 2, we stated that typical depth first pattern mining algorithms do not exhibit super linear speedup. By restructuring computation for smaller tile sizes, and by assigning tiles to different cores, it may be possible to reduce the working set, thus affording the possibility for super linear speedup. We are currently evaluating this possibility by performing detailed simulation of our codes for CMP architectures.

Finally, recent research has given rise to the possibility of hardware-assisted task management[6]. An anticipated outcome of such research is that the overheads associated with task management will be significantly lowered in next generation processors. This fits in well with the fine-grained task partitioning, allocation and queuing schemes presented and evaluated in this paper.

6. Related Work

6.1 Frequent Itemset Mining

Agrawal *et al.* [1] presented *Apriori*, the first efficient algorithm to solve this problem. *Apriori* traverses the itemset search space in breadth-first order. Its efficiency stems from its use of the *anti-monotone* property: *If a size k-itemset is not frequent, then any size $(k+1)$-itemset containing it will not be frequent*. The algorithm first finds all frequent 1-items in the data set, and then iteratively finds all frequent l-itemsets using the frequent $(l-1)$-itemsets discovered previously.

This general level-wise algorithm has been extended in several different forms leading to improvements such as *DHP* [23] and *DIC* [5]. *DHP* uses hashing to reduce the number of candidate itemsets that need to be considered through each data set scan. Furthermore, it progressively prunes the transaction data set as it discovers items that will not be useful during the next data set scan. *DIC* [5] processes the data set in chunks and considers new candidate itemsets as they are discovered through the scanning of a chunk.

Zaki *et al.* proposed *Eclat* [35] and several other algorithms that use equivalence classes to partition the problem into independent subtasks. The use of the vertical data format allows for fast support counting by set intersection. The independent nature of subtasks, coupled with the use of the vertical data format, results in improved I/O efficiency.

Another popular approach to frequent pattern mining is to directly find all maximal [14] or closed frequent itemsets [34], without generating all frequent itemsets in the data set. The benefit of this approach is that maximal or closed frequent itemsets can be used to enumerate all frequent itemsets.

6.2 Graph Mining

Advantages for representing data as graphs has been widely studied [9, 10, 18, 29, 32, 36]. Several serial algorithms exist which enumerate frequent subgraphs. FSG [18] mines for all frequent subgraphs using an APRIORI-like breadth-first strategy. Efficiency is gained by storing intermediate representations, and using vertex invariants to develop a canonical labeling system.

GSpan [31] uses a combination of depth-first and breadth-first trajectories to discover frequent patterns. Unlike previous methods, it only grows graphs with candidates existing in the data set. For canonical labeling, the authors define a mechanism called DFS coding for labeling graphs. This new labeling system reduces the search space considerably. It produced a significant reduction in execution time over previous methods. In addition, gSpan consumes relatively little memory.

Another recent serial graph miner is Gaston [22], developed by Nijssen and Kok. It incorporates embedding lists using an efficient pointer-based structure. This provides an improvement in execution time at the cost of significantly increased memory consumption. The authors build on depth first traversals to specialize labeling systems for all three structures. A limitation of embedding lists is that the parent-child dependencies of the data structure hinder efficient parallelization.

6.3 Parallel Pattern Mining

There have been several research efforts exploring parallel mining for frequent patterns. We only mention the most relevant works here. Agrawal and Shafer [2] presented several *Apriori*-based parallel formulations for frequent pattern mining. They target shared-nothing architectures. *Count Distribution (CD)* parallelizes the frequent itemset discovery process by replicating the candidate generation phase on all processors and parallelizing the counting process. Each iteration is followed by a global reduction operation to assimilate counts. This approach suffers from excessive synchronization and communication overheads. *Data Distribution (DD)* partitions both, the candidates, as well as the data, among the processors. However, this approach requires communication of locally stored transactions between processors, which is an expensive operation. This has been improved upon by Han *et al.* [16] through intelligent data distribution. A Hybrid scheme was also proposed, which partially replicates the candidates between the processors. Parthasarathy *et al.* presented a hash tree-based parallel algorithm for mining frequent patterns on an SMP [27]. This method uses equivalence classes to partition the problem space into independent subtasks. The independent nature of these subtasks minimizes the synchronization overhead, making the mining process scalable.

Wang and Parthasarathy [30] developed a parallel algorithm for their Motif Miner toolkit [25]. Their parallelization strategy cannot be directly applied to the more general graph mining problem.

Our own efforts in sequence and itemset mining have shown us that static load balancing models behave poorly in pattern mining because the time to mine even a single task is highly variable [24]. We expect graph mining to exhibit worse behavior. Some researchers [8] have used random sampling to combat this problem for other frequent pattern algorithms, although it has not been shown to be effective for graph mining, since random sampling may not capture important relations among nodes.

Zaki [33] proposed parallel partitioning schemes for sequence mining. The author illustrates the benefits of dynamic load balancing in shared memory environments. However, a key difference is that estimates for task execution time (used by the author) are far easier to compute in sequence mining than for graph mining. Guralnik and Karypis had similar findings in [15].

7. Conclusion

We have shown that algorithms which incorporate runtime analysis as a controlling mechanism for task allocation and granularity can greatly improve parallel pattern mining performance. In addition, we have illustrated that distributed queuing provides the best mix of sharing and locking for this class of workloads. Finally, we have provided novel algorithmic improvements, designed to increase the efficiency of the underlying memory subsystem, which greatly improve end scalability. These techniques are general purpose to pattern mining; we demonstrate efficacy from two ends of the pattern spectrum, namely frequent itemset mining and frequent substructure mining. The end result is a decrease in mining runtimes by up to a factor of 27 on 32 nodes, which is a major improvement over extant methods. In the near future, we plan to improve upon this by incorporating dynamic state management for shared cache architectures such as future chip multiprocessor systems. As CMP systems are increasing in both availability and popularity, efficient algorithms specifically designed to leverage their strengths become paramount.

Acknowledgments

The majority of the work described in this article was carried out at the Ohio State University and was supported by NSF grants (CAREER-IIS-0347662), (RI-CNS-0403342), and (NGS-CNS-0406386). Aspects of the work described in 3.1 were evaluated when the first author was an intern at Intel Research. The findings and results reported in this work are the authors' own and do not necessarily reflect the views of the Ohio State University, the National Science Foundation, or Intel Corporation.

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. Schafer. Parallel mining of association rules. *IEEE Transactions on Knowledge and Data Engineering*, 1996.

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

[4] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *The Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, 2000.

[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] J. Chen, P. Juang, K. Ko, G. Contreras, D. Penry, R. Rangan, A. Stoler, L.-S. Peh, and M. Martonosi. Hardware-modulated parallelism in chip multiprocessors. *SIGARCH Comput. Archit. News*, 33(4):54–63, 2005.

[7] Y. Chen, L. Yang, and Y. Wang. Incremental mining of frequent xml query patterns. In *Proceedings of the International Conference on Data Mining (ICDM)*, 1999.

[8] 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. ACM Press, 2005.

[9] D. J. Cook, L. B. Holder, G. Galal, and R. Maglothin. Approaches to parallel graph-based knowledge discovery. volume 61, pages 427–446, Orlando, FL, USA, 2001. Academic Press, Inc.

[10] L. Dehaspe, H. Toivonen, and R. D. King. Finding frequent substructures in chemical compounds. In *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining*, pages 30–36. AAAI Press, 1998.

[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] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y. Chen, and P. Dubey. Cache-conscious frequent pattern mining on modern and emerging architectures. In *OSU Technical Report*, volume OSU-CISRC-3/06-TR31, 2006.

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

[14] K. Gouda and M. Zaki. Efficiently mining maximal frequent itemsets. In *Proceedings of the International Conference on Data Mining (ICDM)*, 2001.

[15] V. Guralnik and G. Karypis. Dynamic load balancing algorithms for sequence mining. In *Univeristy of Minnesota Technical Report TR 00-056*, 2001.

[16] E. Han, G. Karypis, and V. Kumar. Scalable parallel mining of association rules. *IEEE Transactions on Knowledge and Data Engineering*, 2000.

[17] 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.

[18] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proceedings of the Internation Conference on Data Mining (ICDM)*, pages 313–320, 2001.

[19] E. Markatos and T. Leblanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. In *IEEE Transactions on Parallel and Distriuted Systems*, 1993.

[20] B. McKay. Practical graph isomorphism. In *Congressus Numerantium*, volume 30, pages 45–87, 1981.

[21] T. Meinl, I. Fischer, and M. Philippsen. Parallel mining for frequent fragments on a shared-memory multiprocessor -results and java-obstacles-. In *LWA 2005 - Beitrge zur GI-Workshopwoche Lernen, Wissensentdeckung, Adaptivitt*, pages 196–201, Saarbrcken, Germany, 2005.

[22] S. Nijssen and J. N. Kok. A quickstart in frequent structure mining can make a difference. In *Proceedings of the 10th International Conference on Knowledge Discovery and Data mining (KDD)*, pages 647–652, New York, NY, USA, 2004. ACM Press.

[23] J. Park, M. Chen, and P. Yu. An effective hash-based algorithm for mining association rules. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 1995.

[24] S. Parthasarathy. Active data mining in a distributed setting. In *University of Rochester Ph.D. Thesis*, 2000.

[25] S. Parthasarathy and M. Coatney. Efficient discovery of common substructures in macromolecules. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, 2002.

[26] 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.

[27] S. Parthasarathy, M. J. Zaki, M. Ogihara, and W. Li. Parallel data mining for association rules on shared-memory systems. In *Knowledge and Information Systems*, volume 3, pages 1–29, 2001.

[28] J. Punin, M. Krishnamoorthy, and M. J. Zaki. Logml – log markup language for web usage mining. In *WEBKDD Workshop: Mining Log Data Across All Customer TouchPoints (with SIGKDD01)*, 2001.

[29] A. Srinivasan, R. King, S. Muggleton, and M. Sternberg. Carcinogenesis predictions using ilp. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297, pages 273–287. Springer-Verlag, 1997.

[30] C. Wang and S. Parthasarathy. Parallel algorithms for mining frequent structural motifs in scientific data. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, 2004.

[31] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 International Conference on Data Mining (ICDM)*, 2002.

[32] X. Yan, X. J. Zhou, and J. Han. Mining closed relational graphs with connectivity constraints. In *Proceedings of the 11th International Conference on Knowledge Discovery and Data mining (KDD)*, 2005.

[33] M. Zaki. Parallel sequence mining on shared-memory machines. In *Journal of Parallel and Distributed Computing*, volume 61, pages 401–426, 2001.

[34] M. Zaki and C. Hsiao. CHARM: An efficient algorithm for closed itemset mining. In *Proceedings of SIAM International Conference on Data Mining (SDM)*, 2002.

[35] M. Zaki, S. Parthasarathy, , M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. pages 283–296, 1997.

[36] M. J. Zaki, V. Nadimpally, D. Bardhan, and C. Bystroff. Predicting protein folding pathways. *Bioinformatics*, 20(1):386–393, 2004.