Parallel Sequence Mining on Shared-Memory Machines

Mohammed J. Zaki

Computer Science Department, Rensselaer Polytechnic Institute, Troy, NY 12180
zaki@cs.rpi.edu
http://www.cs.rpi.edu/ zaki

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

1 Introduction

The sequence mining task is to discover a sequence of attributes, shared across time among a large number of objects in a given database. For example, consider a web access database at a popular site, where an object is a web user and an attribute is a web page. The discovered patterns are the sequences of most frequently accessed pages at that site. This kind of information can be used to restructure the web-site, or to dynamically insert relevant links in web pages based on user access patterns. There are many other domains where sequence mining has been applied, which include discovering customer buying patterns in retail stores, identifying plan failures [1], finding network alarm patterns [2], and so on.

The task of discovering all frequent sequences in large databases is quite challenging. The search space is extremely large. For example, with m attributes there are, in the worst case, $O(m^k)$ potential sequences of length at most k. Fortunately, in practice only a small fraction of all potential sequences are shared among many database objects or transactions, the so-called frequent sequences. Nevertheless, given the search complexity, serial algorithms cannot provide scalability, in terms of the data size and the performance, for large databases. Because there is always this limit to the performance of a single processor, we must rely on parallel multiprocessor systems to fill this role.

Two approaches for utilizing multiple processors have emerged: *distributed memory*, in which each processor has a private memory; and *shared memory*, in

which all processors access common memory. A shared-memory architecture has many desirable properties. Each processor has direct and equal access to all the memory in the system. Parallel programs are easy to implement on such a system. A different approach to multiprocessing is to build a system from many units, each containing a processor and memory. In a distributed memory architecture, each processor has its own local memory that can only be accessed directly by that processor. For a processor to have access to data in the local memory of another processor, a copy of the desired data elements must be sent from one processor to the other, utilizing the message passing programming paradigm. Although a shared memory architecture offers programming simplicity, the finite bandwidth of a common bus can limit scalability. A distributed memory architecture cures the scalability problem by eliminating the bus, but at the cost of programming simplicity. It is possible to combine the best of both the worlds by providing a shared global address space abstraction over physically distributed memory. Such an architecture is called distributed-shared memory (DSM) system. It provides ease of programming, yet retains scalability at the same time. The shared-memory abstraction can be provided in hardware or software.

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

In this paper we present pSPADE, a parallel algorithm for discovering the set of all frequent sequences, targeting shared-memory systems. pSPADE is an asynchronous algorithm, in that it requires no synchronization among processors, except when a load imbalance is detected. For sequence mining on large databases with millions of transactions the problem of I/O minimization becomes paramount. However, most current algorithms are iterative in nature, requiring as many full database scans as the longest frequent sequence, which is clearly very expensive. Some of the methods, especially those using some form of sampling, can be sensitive to the data-skew, which can adversely effect performance. Most approaches also use very complicated internal data structures which have poor locality [3], and add additional space and computation overheads. pSPADE has been designed such that it has good locality and has little false sharing.

The key features of our approach are as follows:

1. We use a *vertical idlist* database format, where we associate with each sequence a list of objects in which it occurs, along with the time-stamps. We show that all frequent sequences can be enumerated via simple temporal idlist intersections.
2. We use a lattice-theoretic approach to decompose the original search space into smaller pieces — the suffix-based classes — which can be processed independently in main-memory. This decomposition is recursively applied within each parent class to produce even smaller classes at the next level,
3. We propose an asynchronous algorithm, where processors work on separate classes at the first level, without any need for sharing or synchronization. To ensure good load balance, we propose a dynamic load balancing scheme, where any free processors joins a busy processor in solving newly formed classes at higher levels.

pSPADE is based on SPADE [4], a sequential algorithm for efficient enumeration of frequent sequences, and thus shares many of its performance features. pSPADE not only minimizes I/O costs by reducing database scans, but also minimizes computational costs by using efficient search schemes. The vertical idlist based approach is also relatively insensitive to data-skew. In fact, idlist skew leads to faster support counting, since the result of an intersection of two lists is always bounded by the size of the *smaller* idlist. An extensive set of experiments is performed on a 12 processor SGI Origin 2000. pSPADE delivers reasonably good speedup, and scales linearly in the database size, and a number of other database parameters.

The rest of the paper is organized as follows: We describe the sequence discovery problem in Section 2 and discuss related work in Section 3. Section 4 describes the serial algorithm, while the design and implementation issues for pSPADE are presented in Section 5. An experimental study is presented in Section 6, and we conclude in Section 7.

2 Sequence Mining

The problem of mining sequential patterns can be stated as follows: Let $\mathcal{I} = \{i_1, i_2, \cdots, i_m\}$ be a set of m distinct attributes, also called *items*. An *itemset* is a non-empty unordered collection of items (without loss of generality, we assume that items of an itemset are sorted in increasing order). All items in an itemset are assumed to occur at the same time. A *sequence* is an ordered list of itemsets. The itemsets in a sequence are ordered according to their associated time-stamp. An itemset i is denoted as $(i_1 i_2 \cdots i_k)$, where i_j is an item. An itemset with k items is called a *k-itemset*. A sequence α is denoted as $(\alpha_1 \mapsto \alpha_2 \mapsto \cdots \mapsto \alpha_q)$, where the sequence *element* α_j is an itemset. A sequence with k items $(k = \sum_j |\alpha_j|)$ is called a *k-sequence*. For example, $(B \mapsto AC)$ is a 3-sequence. An item can occur only once in an itemset, but it can occur multiple times in different itemsets of a sequence.

A sequence $\alpha = (\alpha_1 \mapsto \alpha_2 \mapsto \cdots \mapsto \alpha_n)$ is a *subsequence* of another sequence $\beta = (\beta_1 \mapsto \beta_2 \mapsto \cdots \mapsto \beta_m)$, denoted as $\alpha \preceq \beta$, if there exist integers $i_1 < i_2 <$

$\cdots < i_n$ such that $\alpha_j \subseteq \beta_{i_j}$ for all α_j. For example the sequence $(B \mapsto AC)$ is a subsequence of $(AB \mapsto E \mapsto ACD)$, since the sequence elements $B \subseteq AB$, and $AC \subseteq ACD$. On the other hand the sequence $(AB \mapsto E)$ is not a subsequence of (ABE), and vice versa. We say that α is a proper subsequence of β, denoted $\alpha \prec \beta$, if $\alpha \preceq \beta$ and $\beta \npreceq \alpha$. A sequence is *maximal* if it is not a subsequence of any other sequence.

A *transaction* $\mathcal{T}$ has a unique identifier and *contains* a set of items, i.e., $\mathcal{T} \subseteq \mathcal{I}$. A *customer*, $\mathcal{C}$, has a unique identifier and has associated with it a list of transactions $\{\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_n\}$. Without loss of generality, we assume that no customer has more than one transaction with the same time-stamp, so that we can use the transaction-time as the transaction identifier. We also assume that the list of customer transactions is sorted by the transaction-time. Thus the list of transactions of a customer is itself a sequence $\mathcal{T}_1 \mapsto \mathcal{T}_2 \mapsto \cdots \mapsto \mathcal{T}_n$, called the *customer-sequence*. The database, $\mathcal{D}$, consists of a number of such customer-sequences.

A customer-sequence, $\mathcal{C}$, is said to *contain* a sequence α, if $\alpha \preceq \mathcal{C}$, i.e., if α is a subsequence of the customer-sequence $\mathcal{C}$. The *support* or *frequency* of a sequence, denoted $\sigma(\alpha)$, is the the total number of customers that contain this sequence. Given a user-specified threshold called the *minimum support* (denoted *min_sup*), we say that a sequence is *frequent* if occurs more than *min_sup* times. The set of frequent k-sequences is denoted as $\mathcal{F}_k$.

Given a database $\mathcal{D}$ of customer sequences and *min_sup*, the problem of mining sequential patterns is to find all frequent sequences in the database. For example, consider the customer database shown in figure 1. The database has three items (A, B, C), four customers, and twelve transactions in all. The figure also shows all the frequent sequences with a minimum support of 75% or 3 customers.

3   Related Work

3.1   Serial Algorithms

The problem of mining sequential patterns was introduced in [5]. They also presented three algorithms for solving this problem. The *AprioriAll* algorithm was shown to perform equal to or better than the other two approaches. In subsequent work [6], the same authors proposed the GSP algorithm that outperformed *AprioriAll* by up to 20 times. They also introduced maximum gap, minimum gap, and sliding window constraints on the discovered sequences. Recently, SPADE [4] was shown to outperform GSP by a factor of two in the general case, and by a factor of ten with a pre-processing step. We therefore based pSPADE on our sequential SPADE method.

The problem of finding *frequent episodes* in a sequence of events was presented in [7]. An episode consists of a set of events and an associated partial order over the events. Our definition of a sequence can be expressed as an episode, however their work is targeted to discover the frequent episodes in a single long

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

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

Fig. 1. Original Database

event sequence, while we are interested in finding frequent sequences across many different customer-sequences. They further extended their framework in [8] to discover *generalized episodes*, which allows one to express arbitrary unary conditions on individual episode events, or binary conditions on event pairs. The MEDD and MSDD algorithms [9] discover patterns in multiple event sequences; they explore the rule space directly instead of the sequence space.

The GSP Algorithm Before we proceed further, we need to give some more details on GSP, since it is forms the core of the previous work on parallel sequence mining.

$\mathcal{F}_1 = \{$ frequent 1-sequences $\}$;
for $(k = 2; \mathcal{F}_{k-1} \neq \emptyset; k = k + 1)$ **do**
    C_k = Set of candidate k-sequences;
    for all customer-sequences $\mathcal{E}$ in the database **do**
        Increment count of all $\alpha \in C_k$ contained in $\mathcal{E}$
    $\mathcal{F}_k = \{\alpha \in C_k | \alpha.sup \geq min\_sup\}$;
Set of all frequent sequences = $\bigcup_k \mathcal{F}_k$;

Fig. 2. The GSP Algorithm

GSP makes multiple passes over the database. In the first pass, all single items (1-sequences) are counted. From the frequent items a set of *candidate* 2-sequences are formed. Another pass is made to gather their support. The frequent 2-sequences are used to generate the candidate 3-sequences, and this process is repeated until no more frequent sequences are found. There are two main steps in the algorithm.

1. **Candidate Generation**: Given the set of frequent $(k-1)$-sequences $\mathcal{F}_{k-1}$, the candidates for the next pass are generated by joining $\mathcal{F}_{k-1}$ with itself. A pruning phase eliminates any sequence at least one of whose subsequences is not frequent. For fast counting, the candidate sequences are stored in a *hash-tree*.
2. **Support Counting**: To find all candidates contained in a customer-sequence $\mathcal{E}$, all k-subsequences of $\mathcal{E}$ are generated. For each such subsequence a search is made in the hash-tree. If a candidate in the hash-tree matches the subsequence, its count is incremented.

The GSP algorithm is shown in Figure 2. For more details on the specific mechanisms for constructing and searching hash-trees, please refer to [6] (note: the second iteration is optimized to directly use arrays for counting the support of 2-sequences, instead of using hash trees).

3.2   Parallel Algorithms

While parallel association mining has attracted wide attention [10–16], there has been relatively less work on parallel mining of sequential patterns. Three parallel algorithms based on GSP were presented in [17]. All three approaches partition the datasets into equal sized blocks among the nodes. In NPSPM, the candidate sequences are replicated on all the processors, and each processor gathers local support using its local database block. A reduction is performed after each iteration to get the global supports. Since NPSPM replicates the entire candidate set on each node, it can run into memory overflow problems for large databases. SPSPM partitions the candidate set into equal-sized blocks and assigns each block to a separate processor. While SPSPM utilizes the aggregate memory of the system, it suffers from excessive communication, since each processor's local database has to be broadcast to all other processors to get the global support. HPSPM uses a more intelligent strategy to partition the candidate sequences using a hashing mechanism. It also reduces the amount of communication needed to count the global support. Experiments were performed on an IBM SP2 distributed memory machine. HPSPM was shown to be the best approach.

The main limitation of all these parallel algorithms is that they make repeated passes over the disk-resident database partition, incurring high I/O overheads. Furthermore, the schemes involve exchanging the remote database partitions during each iteration, resulting in high communication and synchronization overhead. They also use complicated hash structures, which entail additional

overhead in maintenance and search, and typically also have poor cache locality [3]. As we shall show in the experimental section, pSPADE is successful in overcoming all these problems.

pSPADE bears similarity to our previous parallel association mining work [16], but it differs in three important respects. First, the itemset search space forms a very small subset of the the sequence search space. Many of the optimizations proposed for generating clique-based partitions of the search space no longer work. The temporal idlist intersections also differ significantly from the non-temporal joins in associations. Second, the association work presented distributed memory algorithms, while pSPADE targets shared-memory systems, the first such study for parallel sequence mining. Finally, pSPADE uses a recursive dynamic load balancing scheme, in contrast to the purely static load balancing scheme used for association mining in [16].

4   The Serial SPADE Algorithm

In this section we describe SPADE [4], a serial algorithm for fast discovery of frequent sequences, which forms the basis for the parallel pSPADE algorithm.

Sequence Lattice SPADE uses the observation that the subsequence relation $\preceq$ defines a partial order on the set of sequences, also called a *specialization relation.* If $\alpha \preceq \beta$, we say that α is more general than β, or β is more specific than α. The second observation used is that the relation $\preceq$ is a *monotone specialization relation* with respect to the frequency $\sigma(\alpha)$, i.e., if β is a frequent sequence, then all subsequences $\alpha \preceq \beta$ are also frequent. The algorithm systematically searches the sequence lattice spanned by the subsequence relation, from the most general to the maximally specific frequent sequences in a breadth/depth-first manner. For example, Figure 3 A) shows the lattice of frequent sequences for our example database.

Support Counting Most of the current sequence mining algorithms [6] assume a *horizontal* database layout such as the one shown in Figure 1. In the horizontal format the database consists of a set of customers (*cid's*). Each customer has a set of transactions (*tid's*), along with the items contained in the transaction. In contrast, we use a *vertical* database layout, where we associate with each item X in the sequence lattice its *idlist*, denoted $\mathcal{L}(X)$, which is a list of all customer (*cid*) and transaction identifiers (*tid*) pairs containing the atom. Figure 3 B) shows the idlists for all the frequent items.

Given the sequence idlists, we can determine the support of any k-sequence by simply intersecting the idlists of any two of its $(k-1)$ length subsequences. In particular, we use the two $(k-1)$ length subsequences that share a common suffix (the generating sequences) to compute the support of a new k length sequence. A simple check on the cardinality of the resulting idlist (actually, the number of distinct *cid*s) tells us whether the new sequence is frequent or not. Figure 3C) shows this process pictorially. It shows the initial vertical database
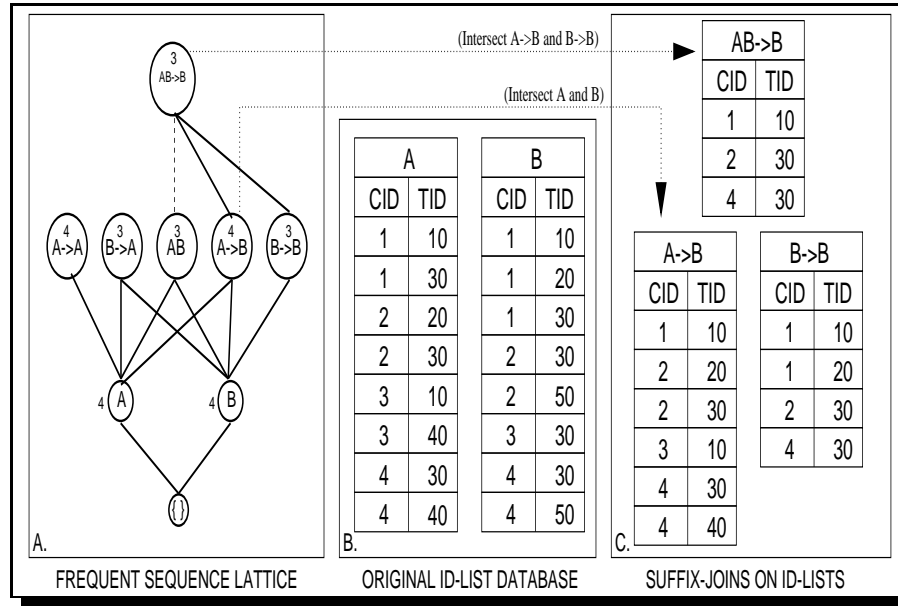
Fig. 3. A. Frequent Sequence Lattice; B. Initial Idlist Database; C. Temporal Idlist Intersections

with the idlist for each item. The intermediate idlist for $A \mapsto B$ is obtained by intersecting the lists of A and B, i.e., $\mathcal{L}(A \mapsto B) = \mathcal{L}(A) \cap \mathcal{L}(B)$. Similarly, $\mathcal{L}(AB \mapsto B) = \mathcal{L}(A \mapsto B) \cap \mathcal{L}(B \mapsto B)$. The temporal intersection is more involved; exact details will be discussed below.

Lattice Decomposition – Suffix-Based Classes If we had enough main-memory, we could enumerate all the frequent sequences by traversing the lattice, and performing temporal intersections to obtain sequence supports. In practice, however, we only have a limited amount of main-memory, and all the intermediate idlists will not fit in memory. SPADE breaks up this large search space into small, independent, manageable chunks which can be processed in memory. This is accomplished via suffix-based equivalence classes. We say that two k length sequences are in the same class if they share a common $k-1$ length suffix. The key observation is that each class is a sub-lattice of the original sequence lattice and can be processed independently. For example, Figure 4A) shows the effect of decomposing the frequent sequence lattice for our example database, by collapsing all sequences with the same 1-length suffix into a single class. There are two resulting suffix classes, namely, $\{[A], [B]\}$, which are referred to as *parent classes*. Each class is independent in the sense that it has complete information for generating all frequent sequences that share the same suffix. For example, if a class $[X]$ has the elements $Y \mapsto X$, and $Z \mapsto X$. The only possible frequent sequences at the next step can be $Y \mapsto Z \mapsto X$, $Z \mapsto Y \mapsto X$, and $(YZ) \mapsto X$.

It should be obvious that no other item Q can lead to a frequent sequence with the suffix X, unless (QX) or $Q \mapsto X$ is also in $[X]$.
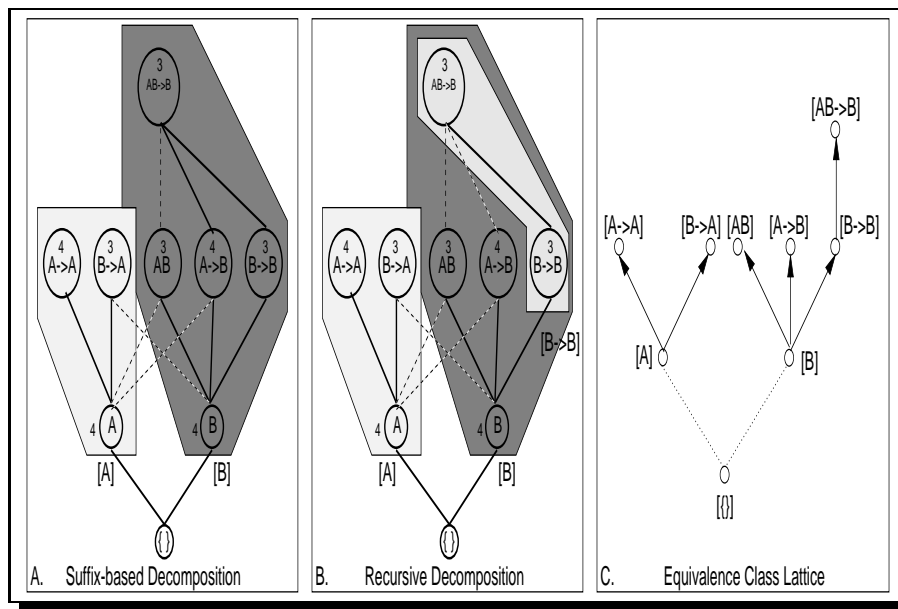

Fig. 4. A) Initial Decomposition: Suffix Length 1, B) Level 2 Decomposition: Suffix Length 2, C) Recursive Decomposition: Class Tree

SPADE recursively decomposes the sequences at each new level into even smaller independent classes. Figure 4B) shows the effect of using 2-length suffixes. If we do this at all levels we obtain a tree of independent classes as shown in Figure 4C). This computation tree is processed in a breadth-first manner, within each parent class. In other words, parent classes are processed one-by-one, but within a parent class we process the new classes in a breadth-first search (BF-S). Figure 5 shows the pseudo-code for the breadth-first search in SPADE. The input to the procedure is a list of classes $PrevL$, along with the idlist for each of their elements. Frequent sequences are generated by intersecting the idlists of all pairs of sequences in each class and checking the cardinality of the resulting idlist against min_sup. The sequences found to be frequent at the current level form classes for the next level $NewL$. This level-wise process is repeated until all frequent sequences have been enumerated. In terms of memory management it is easy to see that we need memory to store intermediate idlists for at most two consecutive levels within a parent class. Once all the frequent sequences for the next level have been generated, the sequences at the current level can be deleted.

SPADE $(min\_sup, \mathcal{D})$:
    $\mathcal{F}_1 = \{$ frequent items or 1-sequences $\}$;
    $\mathcal{F}_2 = \{$ frequent 2-sequences $\}$;
    $\mathcal{C} = \{$ parent classes $C_i = [X_i]\}$;
    for each $C_i \in \mathcal{C}$ **do** *Enumerate-Frequent-Seq(C_i)*;

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

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

Fig. 5. Pseudo-code for SPADE

Constructing Parent Classes The SPADE algorithm performs BFS search for each parent class. Each parent class is constructed from the set of frequent 2-sequences. A sequence of the form $Y \mapsto X$ or YX is added to the suffix class $[X]$. Let $N = |\mathcal{I}|$ be the number of frequent items, and A the average idlist size in bytes. A naive implementation for computing the frequent 2-sequences requires $\binom{N}{2}$ idlist intersections for all pairs of items. The amount of data read is $A \cdot N \cdot (N-1)/2$, which corresponds to around $N/2$ data scans. This is clearly inefficient. Instead of the naive method, we use a preprocessing step to gather the counts of all 2-sequences above a user specified lower bound. Since this information is invariant, it has to be computed once, and the cost can be amortized over the number of times the data is mined. For another method that doesn't require pre-processing, and for additional details on the SPADE algorithm, we refer the reader to [4].

Disk Scans Before processing each of the parent classes from the initial decomposition, all the relevant item idlists for that class are scanned into from disk into memory. All the other frequent sequences are enumerated using temporal joins. If all the initial classes have disjoint set of items, then each item's idlist is scanned from disk only once during the entire frequent sequence enumeration process over all sub-lattices. In the general case there will be some degree of overlap of items among the different sub-lattices. However, only the database portion corresponding to the frequent items will need to be scanned, which can be a lot smaller than the entire database. Furthermore, sub-lattices sharing many common items can be processed in a batch mode to minimize disk access. Thus,

our algorithms will usually requires only a few database scans, in contrast to the current approaches which require as many scans as the longest frequent sequence (this can be reduced somewhat by combining candidates of multiple lengths in later passes).

Temporal Idlist Intersection We now describe how we perform the temporal idlist intersections for two sequences, since this forms the heart of the computation of SPADE, and is crucial in understanding the parallelization strategies.

Given a suffix equivalence class $[S]$, it can contain two kinds of elements: an itemset of the form XS or a sequence of the form $Y \mapsto S$, where X and Y are items, and S is some (suffix) sequence. Let's assume without loss of generality that the itemsets of a class always precede its sequences. To extend the class for the next level it is sufficient to intersect the idlists of all pairs of elements. However, depending on the pairs being intersected, there can be up to three possible resulting frequent sequences:

1. **Itemset vs Itemset**: If we are intersecting XS with YS, then we get a new itemset XYS.
2. **Itemset vs Sequence**: If we are intersecting XS with $Y \mapsto S$, then the only possible outcome is new sequence $Y \mapsto XS$.
3. **Sequence vs Sequence**: If we are intersecting $X \mapsto S$ with $Y \mapsto S$, then there are three possible outcomes: a new itemset $XY \mapsto S$, and two new sequences $X \mapsto Y \mapsto S$ and $Y \mapsto X \mapsto S$. A special case arises when we intersect $X \mapsto S$ with itself, which can only produce the new sequence $X \mapsto X \mapsto S$.

Consider the idlist for the items A and B shown in Figure 3 B). These are taken to be sequence elements $A \mapsto \emptyset$ and $B \mapsto \emptyset$ for the class $[\emptyset]$. To get the idlist for the resultant itemset AB, we need to check for *equality* of cid-tid pairs. In our example, $\mathcal{L}(AB) = \{(1, 10), (1, 30), (2, 20), (4, 30)\}$. It is frequent at 75% minimum support level (i.e., 3 out of 4 customers). Note that support is incremented only once per customer.

To compute the idlist for the sequence $A \mapsto B$, we need to check for a *follows* temporal relationship, i.e., for a given pair (c, t_1) in $\mathcal{L}(A)$, we check whether there exists a pair (c, t_2) in $\mathcal{L}(B)$ with the same cid c, but with $t_2 > t_1$. If this is true, it means that the item B follows the item A for customer c. The resultant idlist for $A \mapsto B$ is shown in Figure 3 C). We call $A \mapsto B$ the *forward* follows intersection. The idlist of $B \mapsto A$ is obtained by reversing the roles of A and B. We call $B \mapsto A$ the *reverse* follows intersection. As a further optimization, we generate the idlists of the (up to) three possible new sequences in just one join.

5   The Parallel pSPADE Algorithm

In this section we describe the design and implementation of the parallel pSPADE algorithm. We begin with a brief review of the SGI Origin architecture.

5.1   SGI Origin 2000

The SGI Origin 2000 machine is a hardware distributed shared memory CC-NUMA (cache-coherent, non-uniform-memory-access) machine, in which shared main memory is distributed amongst the nodes. This shared memory is accessible to every processor in the system. It is also modular and scalable; that is, the system can be increased in size (scaled) by adding node boards in a hypercube topology, and connected by the CrayLink interconnect. Figure 6 shows the configuration of our 12 processor Origin. It also shows what a full 16-processor system would look like. A 'P' denotes a processor; 'N' a node board, containing two processors and some amount of memory; and 'R' a router that routes data between nodes.
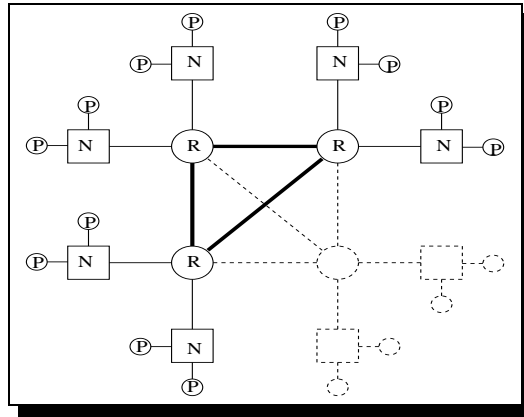

Fig. 6. 12 Processor SGI Origin 2000

5.2   pSPADE: Design and Implementation

pSPADE will be best understood when we imagine the computation as a dynamically expanding irregular tree of independent suffix-based classes, as shown in Figure 7. This example tree represents the search space for the algorithm, with a maximum of five levels. There are three independent parent suffix-based equivalence classes. These are the only classes visible at the beginning of computation. Since we have a shared-memory machine, there is only one copy on disk of the database in the vertical idlist format. It can be accessed by any processor, via a local file descriptor. Given that each class in the tree can be solved independently the crucial issue is how to achieve a good load balance, so that each processor gets an equal amount of work. We would also like to maximize locality and minimize/eliminate cache contention.

There are two main paradigms that may be utilized in the implementation of parallel sequence mining: a *data parallel* approach or a *task parallel* approach.
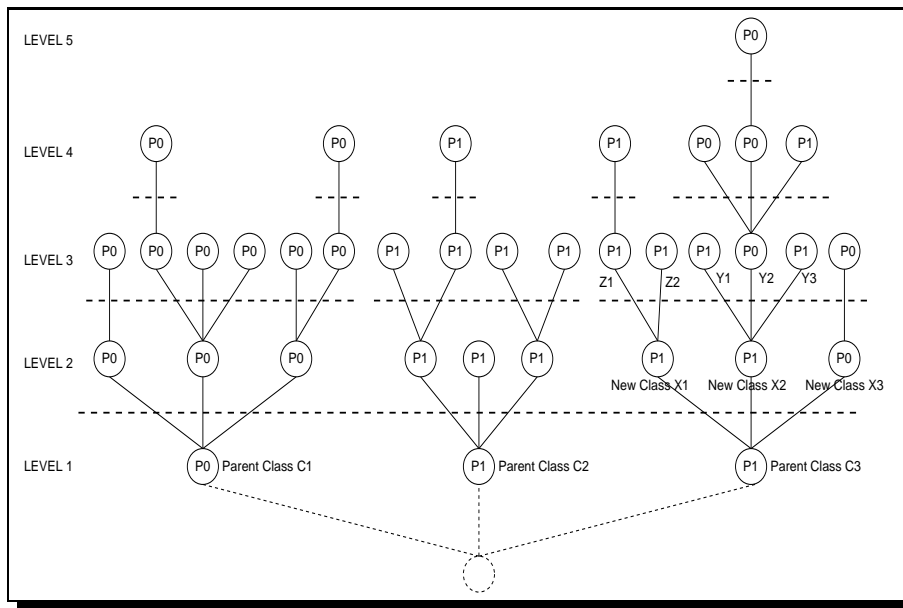
Fig. 7. Dynamic and Irregular Computation Tree of Classes

In data parallelism P processors work on distinct portions of the database, but synchronously process the global computation tree. It essentially exploits intra-class parallelism, i.e., the parallelism available within a class. In task parallelism, the processors share the database, but work on different classes in parallel, asynchronously processing the computation tree. This scheme is thus based on inter-class parallelism.

Data Parallelism As mentioned above, in a data parallelism approach, P processors work on distinct partitions of the database (i.e., idlists), but synchronously process the global computation tree. In other words, we only need to describe how the work of a single node of the computation tree is performed in parallel among all the available processors. Each node corresponds to an equivalence class of frequent sequences, which needs to be expanded to the next level. The main computation within each class is simply the temporal idlist intersections that are performed for all pairs of elements in the class.

Data parallelism can come in two flavors, since we can partition the idlists horizontally or vertically. In horizontal partitioning we split each idlist into blocks and assign these horizontal blocks to processors, while in a vertical partitioning we assign a separate idlists to each processor. The first case corresponds to, what we call, *idlist parallelism*, in which we partition each idlist into P ranges over the customer sequence *cids* (for example, processor 0 is responsible for the *cid* range $0 \cdots l$, processor 1 for range $l + 1 \cdots 2l$, and so on). Each processor is responsible for $1/P$ of the *cids*. The other case corresponds to, what we call,

join parallelism, where each processor picks a sequence (along with its idlist) and performs intersections with the other sequence idlists in the same class, generating new classes for the next level.

Idlist Parallelism There are two ways of implementing the idlist parallelism. In the first method a *single* intersection is performed in parallel among the P processors. Each processor performs the intersection over its *cid* range, and increments support in a shared variable. A barrier synchronization must be performed to make sure that all processors have finished their intersection for the candidate. Finally, based on the support this candidate may be discarded if infrequent or added to the new class if frequent. This scheme suffers from massive synchronization overheads. As we shall see in Section 6, for some values of minimum support we performed around 0.4 million intersections. This scheme will require as many barrier synchronizations.

The other way of implementing idlist parallelism is to use a *level-wise* approach. In other words, at each new level of the computation tree (within a parent class), each processor processes all the new classes at that level, performing intersections for each candidate, but only over its local block. The local supports are stored in a local array to prevent false sharing among processors. After a barrier synchronization signals that all processors have finished processing the current level, a sum-reduction is performed in parallel to determine the global support of each candidate. The frequent sequences are then retained for the next level, and the same process is repeated for other levels until no more frequent sequences are found.

Figure 8 shows the pseudo-code for the single and level-wise idlist data parallelism. The single idlist data parallelism requires modification to the *Get-New-Classes* routine in the SPADE algorithm, by performing each intersection in parallel followed by a barrier (we prefix the modified routine with SID – Single IDlist). The level-wise idlist data parallelism requires modification to the *Enumerate-Frequent-Seq* routine in the SPADE algorithm, by performing local intersection for all classes at the current level, followed by a barrier before the next level can begin (we prefix the modified routine with LID – Level-wise IDlist). Figure 9 depicts the two methods pictorially. For example, in the single idlist method we perform a single intersection, say between items A and B, in parallel; processor P_0 performs intersections on the cid range 1 to 500, while P_1 performs the joins over the cid range 501-1000. Note that even though the ranges are equal, the actual cid's falling in those blocks may be skewed. Figure 9 also shows the level-wise idlist parallelism. In this approach, all processors perform the $\binom{5}{2} + 5 = 15$ possible intersections (i.e., for $AA, AB, AC, AD, AE, BB, BC, \cdots, DE$) in parallel over their cid block, which is then followed by a sum-reduction to get global support.

We implemented the level-wise idlist parallelism and found that it performed very poorly. In fact, we got a speed-down as we increased the number of processors (see Section 6). Even though we tried to minimize the synchronization as much as possible, performance was still unacceptable. Since a candidate's memory cannot be freed until the end of a level, the memory consumption of this

approach is also extremely high. We need to keep the temporary idlists of all newly generated candidates (both infrequent and frequent) since we can't say if a candidate is frequent until all processors have finished the current level. We were thus unable to run this algorithm for low values of minimum support. Also, when the local memory is not sufficient the Origin allocates remote memory for the intermediate idlists, causing a performance hit due to the NUMA architecture.

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

LID-Enumerate-Frequent-Seq($PrevL$):
    while $(PrevL \neq \emptyset)$
        do in parallel for all processors p
            $NewL_p = NewL_p \cup$
                $Get\text{-}New\text{-}Classes$ $(PrevL.item())$;
        $PrevL = PrevL.next()$;
    end while
    barrier;
    $NewL = \bigcup_{p \in P} NewL_p$;
    if $(NewL \neq \emptyset)$ **then**
        $Enumerate\text{-}Frequent\text{-}Seq(NewL)$;

Fig. 8. Single vs. Level-Wise Idlist Data Parallelism

Join Parallelism Join parallelism is based on the vertical partitioning of the idlists among processors. Each processor performs intersections for different sequences within the same class. Once the current class has been expanded by one level, the processors must synchronize, before moving on to the next class. Figure 9 shows how join parallelism works. P_0 gets the items A, C, and E, and is responsible for generating and testing all candidates which have those items as a prefix (i.e., the candidates $AA, AB, AC, AD, AE, CC, CD, CE$, and EE). P_1 on the other hand is responsible for all candidates with the prefix B or D (i.e., BB, BC, BD, BE, DD, and DE). While we have not implemented this approach, we believe that it will fare no better than idlist parallelism. The reason is that it requires one synchronization per class, which is better than the sin-
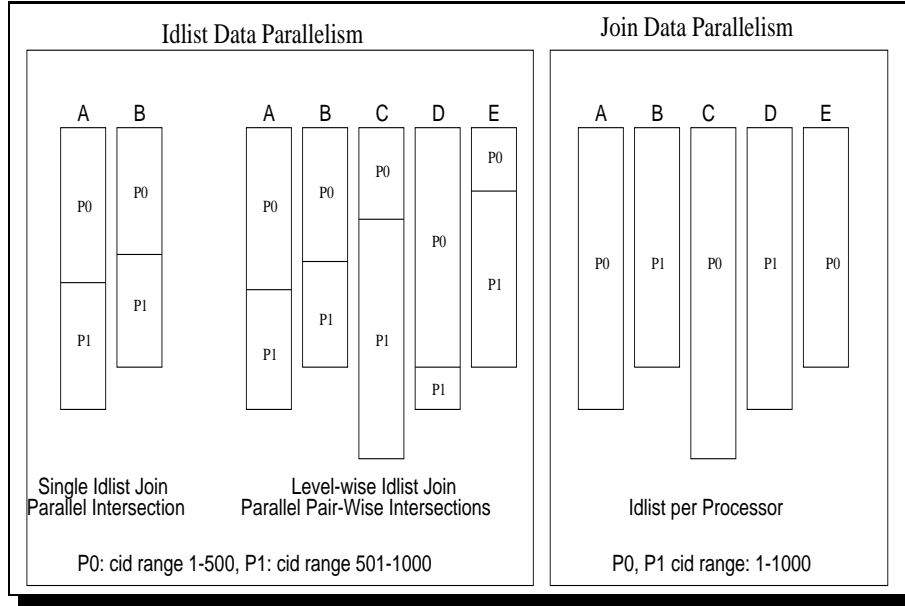
Fig. 9. Idlist (Single and Level-wise) vs. Join Data Parallelism (2 processors, cid range 1-1000)

gle candidate idlist parallelism, but still much worse than the level-wise idlist parallelism, since there can be many classes.

Task Parallelism In task parallelism all processors have access to one copy of the database, but they work on separate classes. We present a number of load balancing approaches starting with a static load balancing scheme and moving on to a more sophisticated dynamic load balancing strategy. It is important to note that we use a breadth first search for frequent sequence enumeration within each parent class, but the parent classes themselves are scheduled independently for good load balance.

Static Load Balancing (SLB) Let $\mathcal{C} = \{C_1, C_2, C_3\}$ represent the set of the parent classes at level 1 as shown in Figure 7. We need to schedule the parent classes among the processors in a manner minimizing load imbalance. In our approach an entire parent class is scheduled on one processor. Load balancing is achieved by assigning a weight to each parent equivalence class based on the number of elements in the class. Since we have to consider all pairs of items for the next iteration, we assign the weight $W_i^1 = \binom{|C_i|}{2}$ to the class C_i. Once the weights are assigned we generate a schedule using a greedy heuristic. We sort the classes on the weights (in decreasing order), and assign each class in turn to the least loaded processor, i.e., one having the least total weight at that point. Ties are broken by selecting the processor with the smaller identifier. These steps

are done concurrently on all the processors since all of them have access to $\mathcal{C}$. We also studied the effect of other heuristics for assigning class weights, such as $W_i^2 = \sum_j |\mathcal{L}(A_j)|$ for all items A_j in the class C_i. This cost function gives each class a weight proportional to the sum of the supports of all the items. We also tried a cost function that combines the above two, i.e., $W_i^3 = \binom{|C_i|}{2} \cdot \sum_j |\mathcal{L}(A_j)|$. We did not observe any significant benefit of one weight function over the other, and decided to use W^1.

SLB ($min\_sup, \mathcal{D}$):
  $\mathcal{C} = \{$ parent classes $C_i = [X_i]\}$;
  Sort-on-Weight($\mathcal{C}$);
  for all $C_i \in \mathcal{C}$ **do** //create work Queue
    $P_j = $ *Proc-with-Min-Weight()*;
    $Q_{P_j} = Q_{P_j} \cup C_i$;
  for all processors P_j
    for all classes $C_i \in Q_{P_j}$ **do** *Enumerate-Frequent-Seq*(C_i);

Fig. 10. The SLB (Static Load Balancing) Algorithm

Figure 10 shows the pseudo-code for the SLB algorithm. We schedule the parent classes on different processors based on the class weights. Once the parent classes have been scheduled, the computation proceeds in a purely asynchronous manner since there is never any need to synchronize or share information among the processors. If we apply W^1 to the class tree shown in Figure 7, we get $W_1^1 = W_2^1 = W_3^1 = 3$. Using the greedy scheduling scheme on two processors, P_0 gets the parent classes C_1 and C_3, and P_1 gets the parent class C_2. The two nodes process these classes in a BFS manner. We immediately see that SLB suffers from load imbalance, since after processing C_1, P_0 will be busy working on C_3, while after processing C_2, P_1 has no more work. The main problem with SLB is that, given the irregular nature of the computation tree there is no way of accurately determining the amount of work (i.e., the number of frequent sequences that might be generated from it) per class statically.

Inter-Class Dynamic Load Balancing (CDLB) To get better load balancing we can utilize inter-class dynamic load balancing. Instead of a static or fixed class assignment of SLB, we would like each processor to dynamically pick a new parent class to work on from the list of parent classes not yet processed.

We also make use of the class weights in the CDLB approach. First, we sort the parent classes in decreasing order of their weight. This forms a logical central task queue of independent classes. Each processor atomically grabs one class from this logical queue. It processes the class completely and then grabs the next available class. This is essentially a self-scheduling scheme [18]. Note that each class usually has a non-trivial or coarse amount of work, so we don't have to worry about contention among processors to acquire new tasks. Since

classes are sorted on their weights, processors first work on large classes before tackling smaller ones, which helps to achieve a greater degree of load balance. The pseudo-code for CDLB algorithm appears in Figure 11. The *compare-and-swap* (CAS) is an atomic primitive on the Origin. It compares *classid* with i. If they are equal it replaces *classid* with $i + 1$, returning a 1, else it returns a 0. The use of CAS ensures that processors acquire separate classes to work on.

CDLB $(min\_sup, \mathcal{D})$:
    $\mathcal{C} = \{$ parent classes $C_i = [X_i]\}$;
    Sort-on-Weight$(\mathcal{C})$;
    shared int classid=0;
    for each processor P_j **do in parallel**
        for $(i = 0; i < |\mathcal{C}|; i + +)$
            if (*compare_and_swap* $(classid, i, i + 1)$)
                Enumerate-Frequent-Seq(C_i);

Fig. 11. The CDLB (Dynamic Load Balancing) Algorithm

If we apply CDLB to our example computation tree in Figure 7, we might expect a scenario as follows: In the beginning P_1 grabs C_1, and P_0 acquires C_2. Since C_2 has less work, P_0 will grab the next class C_3 and work on it. Then P_1 becomes free and finds that there is no more work, while P_0 is still busy. For this example, CDLB did not buy us anything over SLB. However, when we have a large number of parent classes CDLB has a clear advantage over SLB, since a processor grabs a new class only when it has processed its current class. This way only the free processors will acquire new classes, while others continue to process their current class, delivering good processor utilization. We shall see in Section 6 that CDLB can provide up to 40% improvement over SLB. We should reiterate that the processing of classes is still asynchronous. For both SLB and CDLB, false sharing doesn't arise, and all work is performed on local memory, resulting in good locality.

Recursive Dynamic Load Balancing (RDLB) While CDLB improves over SLB by exploiting dynamic load balancing, it does so only at the inter-class level, which may be too coarse-grained to achieve a good workload balance. RDLB addresses this by exploiting both inter-class and intra-class parallelism.

To see where the intra-class parallelism can be exploited, let's examine the behavior of CDLB. As long as there are more parent classes remaining, each processor acquires a new class and processes it completely using BFS search. If there are no more parent classes left, the free processors are forced to idle. The worst case happens when $P-1$ processors are free and only one is busy, especially if the last class has a deep computation tree (although we try to prevent this case from happening by sorting the classes, so that the classes predicted to be small are at the end, it can still happen). We can fix this problem if we provide

a mechanism for the free processors to join the busy ones. We accomplish this by recursively applying the CDLB strategy at each new level, but only if there is some free processor waiting for more work. Since each class is independent, we can treat each class at the new level in the same way we treated the parent classes, so that different processors can work on different classes at the new level.

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

pSPADE (min_sup, D):
4.     GlobalQ = C = { parent classes Cᵢ = [Xᵢ]};
5.     Sort-on-Weight(C);
6.     Process-GlobalQ();
7.     FreeCnt + +;
8.     while (FreeCnt ≠ P)
9.        if (GlobalFlg) then
10.            FreeCnt − −; Process-GlobalQ(); FreeCnt + +;

Process-GlobalQ():
11.      shared int classid = 0;
12.         parallel for (i = 0; i < GlobalQ.size(); i + +)
13.            if (compare_and_swap (classid, i, i + 1))
14.               RDLB-Enumerate-Frequent-Seq(Cᵢ);
15.      GlobalFlg = 0;

RDLB-Enumerate-Frequent-Seq(PrevL):
16.      for (; PrevL ≠ ∅; PrevL = PrevL.next())
17.         if (FreeCnt > 0) then
18.            Add-to-GlobalQ(PrevL.next()); GlobalFlg = 1;
19.         NewL = NewL ∪ Get-New-Classes (PrevL.item());
20.      if (NewL ≠ ∅) then RDLB-Enumerate-Frequent-Seq(NewL);
```

Fig. 12. The pSPADE Algorithm (using RDLB)

Figure 12 shows the pseudo-code for the final pSPADE algorithm, which uses the recursive dynamic load balancing (RDLB) scheme. We start with the parent classes and insert them in the global class list, *GlobalQ*. Each processor atomically acquires classes from this list until all parent classes have been taken, similar to the CDLB approach (*Process-GlobalQ()* on line 6 is the same as the main loop in CDLB). Note that each parent class is processed in a BFS manner.

As each processor finishes its portion of the parent classes, and no more parent classes are left, it increments the shared variable *FreeCnt*, and waits for more work. When a processor is processing the classes at some level, it periodically checks if there are any free processors (line 17). If so, it keeps one class for itself,

and inserts the remaining classes at that level ($PrevL$) in $GlobalQ$, emptying $PrevL$ in the process, and sets $GlobalFlg$. This processor continues working on the classes ($NewL$) generated before a free processor was detected. Note that all idlist intersections are performed in the routine *Get-New-Classes()* (as shown in Figure 5).

When a waiting processor sees that there is more work (i.e., $GlobalFlg = 1$), it starts working on the classes in $GlobalQ$. Finally, when there is no more work in the global queue, $FreeCnt$ equals the number of processors P, and the computation stops. To reiterate, any class inserted into the global queue is treated as a parent class, and is processed in a purely breadth-first manner. If and when a free processor is detected, a busy processor adds all classes on its current level into the global queue for shared processing.

Let's illustrate the above algorithm by looking at the computation tree in Figure 7. The nodes are marked by the processors that work on them. First, at the parent class level, P_0 acquires C_1, and P_1 acquires C_2. Since C_2 is smaller, P_1 grabs class C_3, and starts processing it. It generates three new classes at the next level, $NewL = \{X_1, X_2, X_3\}$, which becomes $PrevL$ when P_1 starts the next level. Let's assume that P_1 finishes processing X_1, and inserts classes Z_1, Z_2 in the new $NewL$.

In the meantime, P_0 becomes free. Before processing X_2, P_1 notices in line 17, that there is a free processor. At this point P_1 inserts X_3 in $GlobalQ$, and empties $PrevL$. It then continues to work on X_2, inserting Y_1, Y_2, Y_3 in $NewL$. P_0 sees the new insertion in $GlobalQ$ and start working on X_3 in its entirety. P_0 meanwhile starts processing the next level classes, $\{Z_1, Z_2, Y_1, Y_2, Y_3\}$. If at any stage it detects a free processor, it will repeat the procedure described above recursively (i.e., inserting remaining classes in $GlobalQ$). Figure 7 shows a possible execution sequence for the class C_3. It can be seen that RDLB tries to achieve as good a load balance as possible by keeping all processors busy.

The RDLB scheme of pSPADE preserves the good features of CDLB, i.e., it dynamically schedules entire parent classes on separate processors, for which the work is purely local, requiring no synchronization, and exploiting only inter-class parallelism so far. Intra-class parallelism is required only for a few (hopefully) small classes towards the end of the computation. We simply treat these as new parent classes, and schedule each class on a separate processor. Again no synchronization is required except for insertions and deletions from $GlobalQ$. In summary, computation is kept local to the extent possible, and synchronization is done only if a load imbalance is detected.

6   Experimental Results

In this section we present the parallel performance of pSPADE. Experiments were performed on a 12 processor SGI Origin 2000 machine at RPI, with 195 MHz R10000 MIPS processors, 4MB of secondary cache per processor, 2GB of main memory, and running IRIX 6.5. The databases were stored on an attached

7GB disk in flat-files. Since there is only one I/O node in our setup, all disk I/O is serial.

Dataset	C	T	S	I	D	Size
C10T5S4I1.25D1M	10	5	4	1.25	1M	320MB
C10T5S4I2.5D1M	10	5	4	2.5	1M	320MB
C20T2.5S4I1.25D1M	20	2.5	4	1.25	1M	440MB
C20T2.5S4I2.5D1M	20	2.5	4	2.5	1M	440MB
C20T2.5S8I1.25D1M	20	5	8	1.25	1M	640MB
C20T5S8I2D1M	20	5	8	2	1M	640MB
C5T2.5S4I1.25DxM	5	2.5	4	1.25	1M-10M	110MB-1.1GB

Table 1. Synthetic Datasets

Dataset	MinSup	# FreqSeq	Time(P=1)	#Seq/Time
C10T5S4I1.25D1M	0.25%	96344	379.7s	254
C10T5S4I2.5D1M	0.33%	180381	625.5s	289
C20T2.5S4I1.25D1M	0.25%	67291	270.3s	249
C20T2.5S4I2.5D1M	0.25%	80648	240.4s	335
C20T2.5S8I1.25D1M	0.33%	55484	236.9s	234
C20T5S8I2D1M	0.5%	179999	1200.8s	150

Table 2. Sequential Time and Number of Frequent Sequences

Synthetic Datasets: We used the publicly available dataset generation code from the IBM Quest data mining project [19]. These datasets mimic real-world transactions, where people buy a sequence of sets of items. Some customers may buy only some items from the sequences, or they may buy items from multiple sequences. The customer sequence size and transaction size are clustered around a mean and a few of them may have many elements. The datasets are generated using the following process. First N_I maximal itemsets of average size I are generated by choosing from N items. Then N_S maximal sequences of average size S are created by assigning itemsets from N_I to each sequence. Next a customer of average C transactions is created, and sequences in N_S are assigned to different customer elements, respecting the average transaction size of T. The generation stops when D customers have been generated. Like [6] we set $N_S = 5000$, $N_I = 25000$ and $N = 10000$. Table 1 shows the datasets with their parameter settings. We refer the reader to [5] for additional details on the dataset generation.

Table 2 shows, for the different datasets, the minimum support used in the experiments reported below, the total number of frequent sequences found, the serial time, and the number of frequent sequences enumerated per second (note: the number of intersections performed is 2-3 times higher). The distribution of frequent sequences as a function of length is plotted in Figure 13. The figure also shows the total number of frequent sequences obtained and the total number of joins performed. The number of joins corresponds to the total number of candidates evaluated during the course of the algorithm.
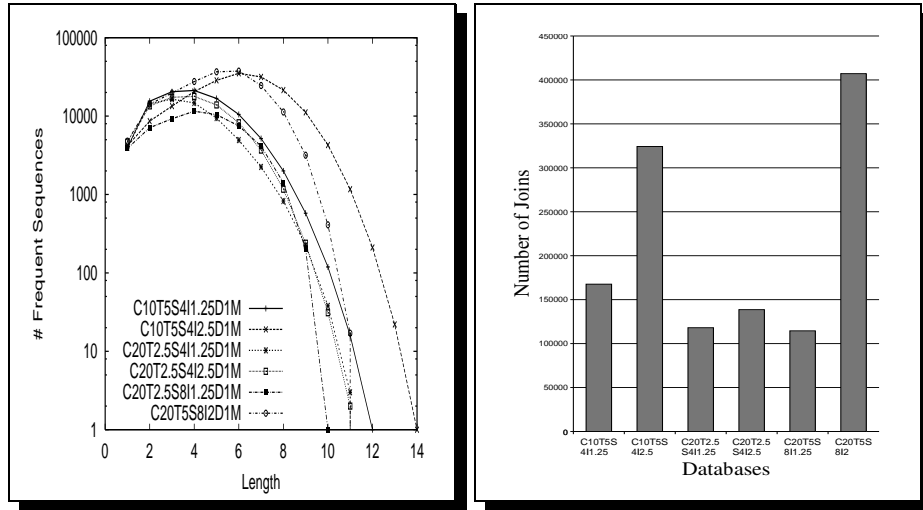
Fig. 13. Number of Frequent Sequences and Candidate Joins

6.1   Serial Performance

The performance of SPADE, the serial version of pSPADE, was studied in [4], and it was compared against GSP [6]. It was shown that SPADE outperforms GSP by more than an order of magnitude if we pre-process the data and store the supports of all frequent 2-sequences above a minimum threshold. The performance comparison of SPADE vs. GSP is shown in Figure 14.

There are several reasons why SPADE outperforms GSP:

1. SPADE uses only simple temporal join operation on idlists. As the length of a frequent sequence increases, the size of its idlist decreases, resulting in very fast joins.
2. No complicated hash-tree structure is used, and no overhead of generating and searching of customer subsequences is incurred. These structures typically have very poor locality [3]. On the other hand SPADE has good locality, since a join requires only a linear scan of two lists.
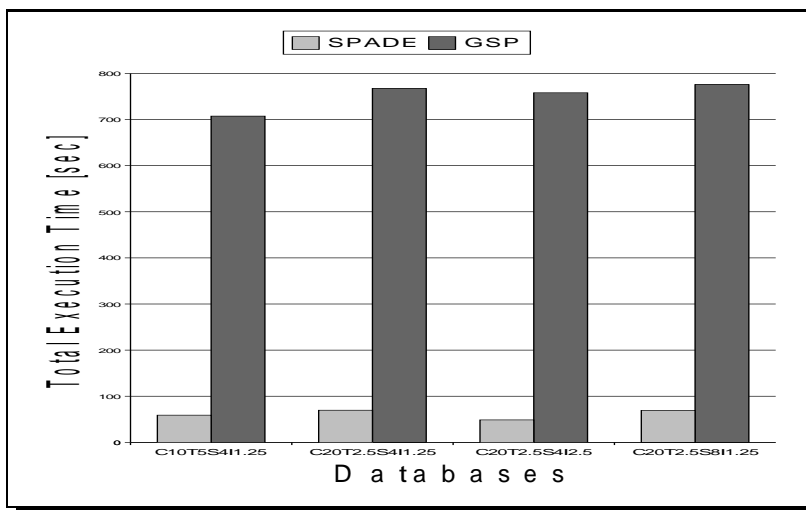
Fig. 14. Serial Performance: SPADE vs. GSP (0.25% Minimum Support; D200K)

3. As the minimum support is lowered, more and larger frequent sequences are found. GSP makes a complete dataset scan for each iteration. SPADE on the other hand restricts itself to only a few scans. It thus cuts down the I/O costs.

These benefits of SPADE carry over to pSPADE. For these reasons we chose not to parallelize GSP for comparison against pSPADE. It should be noted that it is possible to optimize GSP further to reduce the number of database scans by generating candidates of multiple lengths at the same time (if memory permits). However, the base GSP, as described in [6] does not do this.

6.2   Parallel Performance

Data vs. Task Parallelism We first present the results for the level-wise idlist data parallel algorithm we described in Section 5.2. Figure 15A) shows the results for four databases on 1, 2, and 4 processors. We find that the data parallel algorithm performs very poorly, resulting in a speed-down with more processors. The level-wise approach does well initially when the number of tree nodes or classes is relatively few. However, as computation progresses more and more classes are generated and consequently more and more barriers are performed. In fact there are almost as many classes as there are frequent itemsets, requiring as many barriers. For example, for the $C20T2.5S4I2.5D1M$ dataset, the data parallel approach may have performed around 80648 barriers. Since data parallel approach doesn't perform well, we only concentrate on task parallelism in the remainder of this section.

Static vs. Dynamic Load Balancing We now present results on the effect of dynamic load balancing on the parallel performance. Figure 15B) shows the

performance of pSPADE using 8 processors on the different databases under static load balancing (SLB), inter-class dynamic load balancing (CDLB), and the recursive dynamic load balancing (RDLB). We find that CDLB delivers more than 22% improvement over SLB in most cases, and ranges from 7.5% to 38% improvement. RDLB delivers an additional 10% improvement over CDLB in most cases, ranging from 2% to 12%. The overall improvement of using RDLB over SLB ranges from 16% to as high as 44%. Thus our load balancing scheme is extremely effective. All results reported below use the recursive dynamic load balancing scheme.
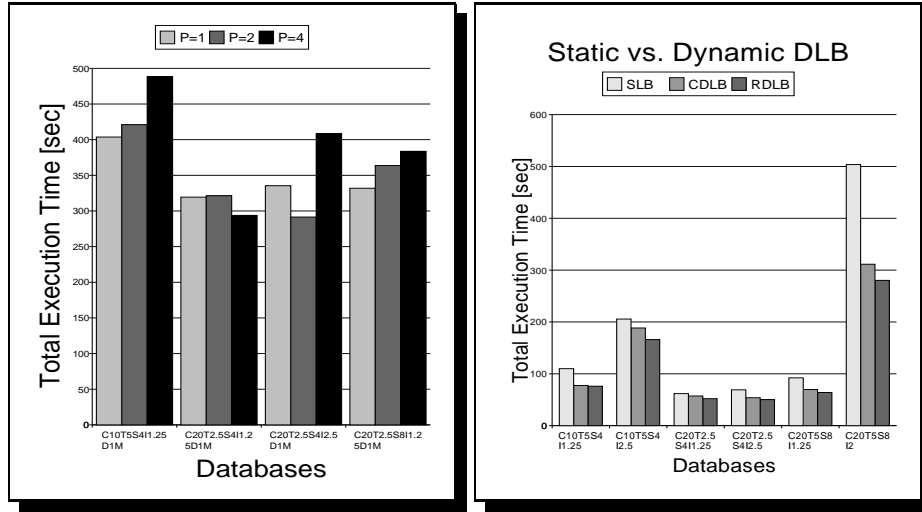

Fig. 15. A) Level-Wise Idlist Data Parallelism, B) Effect of Load Balancing

Parallel Time and Speedup Figure 16 shows the total execution time and the speedup charts for each database using the minimum support values shown in Table 1. We obtain near perfect speedup for 2 processors, ranging as high as 1.91. On 4 processors, we obtained a maximum of 3.2, on 8 processors the maximum was 5.6, and on 12 the maximum speedup was 7.2. As these charts indicate, pSPADE achieves relatively good speedup performance. However, the speedup on C20T5S8I2D1M was not as good. If one looks at the distribution of the frequent sequence lengths for C20T5S8I2D1M in Figure 13 we see that it has many more large frequent sequences compared to other datasets, and has longer idlist sizes as well. Many frequent items imply that there is more overlap of items among the classes, and along with longer idlists this causes more disk reads. In itself this is not a problem, but since the our SGI Origin system only supports serial I/O, this results in increased disk contention, which in turn limits the speedup possible for this dataset. The serial I/O is also one of the causes preventing us from achieving better speedups on other datasets. The other reason
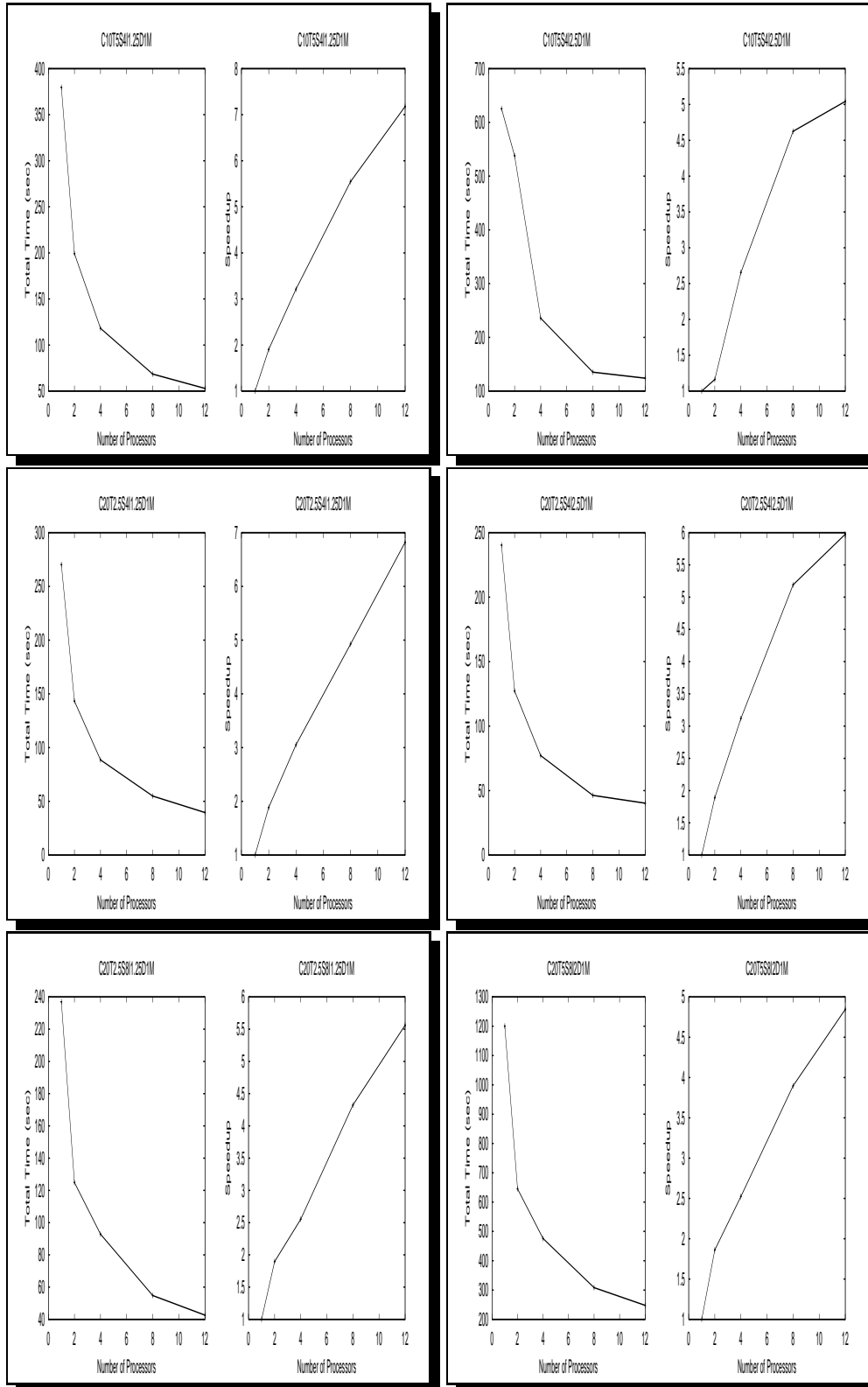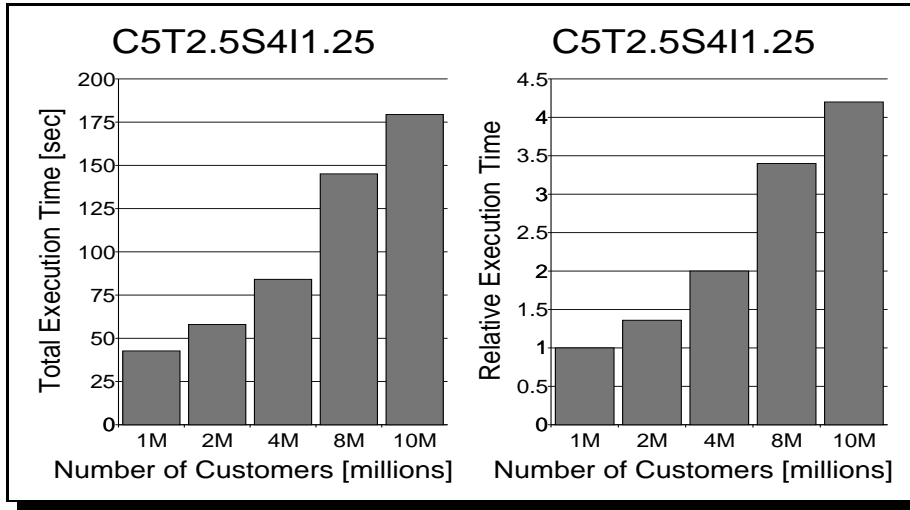
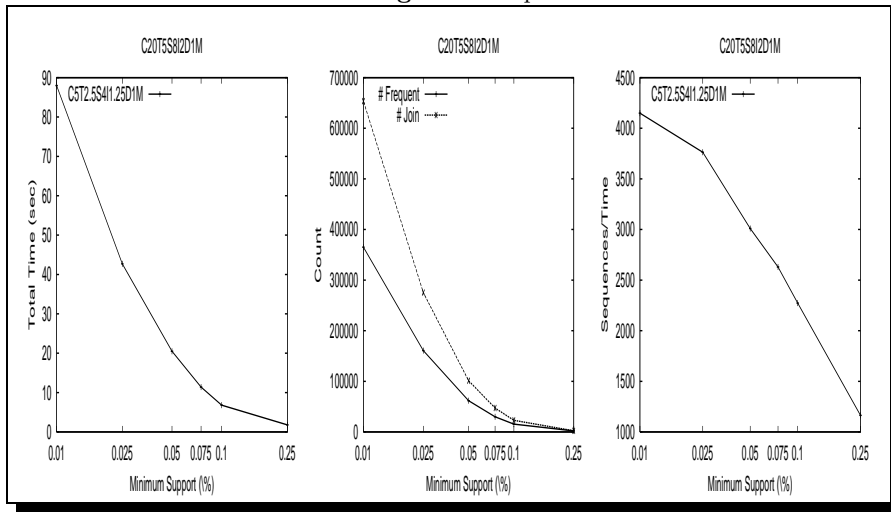**Fig. 16.** pSPADE Parallel Performance

Fig. 17. Sizeup

Fig. 18. Effect of Minimum Support ($C5T2.5S4I1.25D1M$)

is that beyond 8 processors, there isn't enough work for 12 processors, i.e., the computation to overhead (class partitioning, disk contention, etc.) ratio is small. Furthermore, while we do try to schedule disjoint classes on different processors, we have made no attempt to fine-tune the affinity scheduling of threads and the idlists accessed. Since the Origin is has NUMA architecture, there is further scope for performance tuning by allocating groups of related classes to processors that are topologically close, or at least among the two processors on the same node board (see Figure 6).

6.3   Scaleup

Figure 17 shows how pSPADE scales up as the number of customers is increased ten-fold, from 1 million to 10 million (the number of transactions is increased from 5 million to 50 million, respectively). The database size goes from 110MB to 1.1GB. All the experiments were performed on the C5T2.5S4I1.25 dataset with a minimum support of 0.025%. Both the total execution time and the normalized time (with respect to 1M) are shown. It can be seen that while the number of customers increases ten-fold, the execution time goes up by a factor of less than 4.5, displaying super-linear scaleup.

Finally, we study the effect of changing minimum support on the parallel performance, shown in Figure 18. We used 8 processors on C5T2.5S4I1.25D1M dataset. The minimum support was varied from a high of 0.25% to a low of 0.01%. Figure 18 shows the number of frequent sequences discovered and the number of joins performed (candidate sequences) at the different minimum support levels. It also shows the number of frequent sequences enumerated per second. Running time goes from 6.8s at 0.1% support to 88s at 0.01% support, a time ratio of 1:13 vs. a support ratio of 1:10. At the same time the number of frequent sequences goes from 15454 to 365132 (1:24), and the number of joins from 22973 to 653596 (1:29). The number of frequent sequences are, in general, not linear with respect to the minimum support. In the worst case, the number of sequences increase exponentially with decreasing support. However, it appears that for the range of support values we looked at the execution time is near-linear. It is interesting to note that the efficiency of pSPADE increases with decreasing support, i.e., it lists more frequent sequences per second on lower support values.

7   Conclusions

In this paper we presented pSPADE, a new parallel algorithm for fast mining of sequential patterns in large databases. We carefully considered the various parallel design alternatives before choosing the best strategy for pSPADE. These included data parallel approaches like idlist parallelism (single vs. level-wise) and join parallelism. In the task parallel approach we considered different load balancing schemes such as static, dynamic and recursive dynamic. We adopted the recursive dynamic load balancing scheme for pSPADE, which was designed to maximize data locality and minimize synchronization, by allowing each processor to work on disjoint classes. Finally, the scheme minimizes load imbalance by exploiting both inter-class and intra-class parallelism. An extensive set of experiments was been conducted on the SGI Origin CC-NUMA shared memory system, to show that pSPADE has good speedup and excellent scaleup properties.

This work opens several research opportunities, which we plan to address in future work:

1. pSPADE works on the assumption that each class and its intermediate idlists fit in main memory. The mean memory utilization of pSPADE is less than

1% of the database size, but the maximum usage may be as high as 10% [16]. This means that on our Origin system, we can handle around 20GB datasets. One solution for handling larger datasets is to write intermediate idlists to disk when we exceed memory. This requires minimal modification to the pSPADE. However, we need to consider the case where even a single idlist may not fit in memory. In this case we bring in the portion of the two idlists that fit in memory and perform joins on the memory-resident portions, repeating the process until the two list have been joined completely. We plan to implement these techniques in the future.

2. Extending pSPADE to run on CLUMPS or clusters of SMP machines, which are becoming increasingly popular. We could utilize pSPADE on each SMP node, while message passing would be required for load balancing among nodes.

3. pSPADE uses only simple intersection operations, and is thus ideally suited for direct integration with a DBMS. We plan to implement pSPADE directly on top of a parallel DBMS.

4. Extending pSPADE for parallel discovery of *quantitative* and *generalized* sequences – where the quantity of items bought is also considered, and where we introduce time gap constraints, sliding windows, and impose a taxonomy on the items, respectively.

References

1. Zaki, M.J., Lesh, N., Ogihara, M.: PLANMINE: Sequence mining for plan failures. In: 4th Intl. Conf. Knowledge Discovery and Data Mining. (1998)
2. Hatonen, K., Klemettinen, M., Mannila, H., Ronkainen, P., Toivonen, H.: Knowledge discovery from telecommunication network alarm databases. In: 12th Intl. Conf. Data Engineering. (1996)
3. Parthasarathy, S., Zaki, M.J., Li, W.: Memory placement techniques for parallel association mining. In: 4th Intl. Conf. Knowledge Discovery and Data Mining. (1998)
4. Zaki, M.J.: Efficient enumeration of frequent sequences. In: 7th Intl. Conf. on Information and Knowledge Management. (1998)
5. Agrawal, R., Srikant, R.: Mining sequential patterns. In: 11th Intl. Conf. on Data Engg. (1995)
6. Srikant, R., Agrawal, R.: Mining sequential patterns: Generalizations and performance improvements. In: 5th Intl. Conf. Extending Database Technology. (1996)
7. Mannila, H., Toivonen, H., Verkamo, I.: Discovering frequent episodes in sequences. In: 1st Intl. Conf. Knowledge Discovery and Data Mining. (1995)
8. Mannila, H., Toivonen, H.: Discovering generalized episodes using minimal occurences. In: 2nd Intl. Conf. Knowledge Discovery and Data Mining. (1996)
9. Oates, T., Schmill, M.D., Jensen, D., Cohen, P.R.: A family of algorithms for finding temporal structure in data. In: 6th Intl. Workshop on AI and Statistics. (1997)
10. Agrawal, R., Shafer, J.: Parallel mining of association rules. IEEE Trans. on Knowledge and Data Engg. **8** (1996) 962–969
11. Cheung, D., Ng, V., Fu, A., Fu, Y.: Efficient mining of association rules in distributed databases. In: IEEE Trans. on Knowledge and Data Engg. (1996) 8(6):911–922

12. Han, E.H., Karypis, G., Kumar, V.: Scalable parallel data mining for association rules. In: ACM SIGMOD Conf. Management of Data. (1997)
13. Park, J.S., Chen, M., Yu, P.S.: Efficient parallel data mining for association rules. In: ACM Intl. Conf. Information and Knowledge Management. (1995)
14. Shintani, T., Kitsuregawa, M.: Hash based parallel algorithms for mining association rules. In: 4th Intl. Conf. Parallel and Distributed Info. Systems. (1996)
15. Zaki, M.J., Ogihara, M., Parthasarathy, S., Li, W.: Parallel data mining for association rules on shared-memory multi-processors. In: Supercomputing'96. (1996)
16. Zaki, M.J., Parthasarathy, S., Ogihara, M., Li, W.: Parallel algorithms for fast discovery of association rules. Data Mining and Knowledge Discovery: An International Journal **1(4):343-373** (1997)
17. Shintani, T., Kitsuregawa, M.: Mining algorithms for sequential patterns in parallel: Hash based approach. In: Pacific-Asia Conf. on Knowledge Discovery and Data Mining. (1998)
18. Tang, P., Yew, P.C.: Processor self-scheduling for multiple nested parallel loops. In: International Conference On Parallel Processing. (1986)
19. IBM: http://www.almaden.ibm.com/cs/quest/syndata.html. (Quest Data Mining Project, IBM Almaden Research Center, San Jose, CA 95120)