Parallel Sequence Mining on Shared-Memory Machines

Mohammed J. Zaki
Computer Science Dept., Rensselaer Polytechnic Institute, Troy, NY 12180

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 scaleup results.

1 Introduction

The sequence mining task is to discover a sequence of attributes (items), shared across time among a large number of objects (transactions) in a given database. 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 $O(m^k)$ potentially frequent sequences of length at most k. Clearly, sequential algorithms cannot provide scalability, in terms of the data size and the performance, for large databases. We must therefore rely on parallel multiprocessor systems to fill this role.

Previous work on parallel sequence mining has only looked at distributed-memory machines [Shintani and Kitsuregawa, 1998]. In this paper we look at shared-memory systems, which are capable of delivering high performance for low to medium degree of parallelism at an economically attractive price. SMP machines are the dominant types of parallel machines currently used in industry. Individual nodes of parallel distributed-memory machines are also increasingly being designed to be SMP nodes.

Our 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.

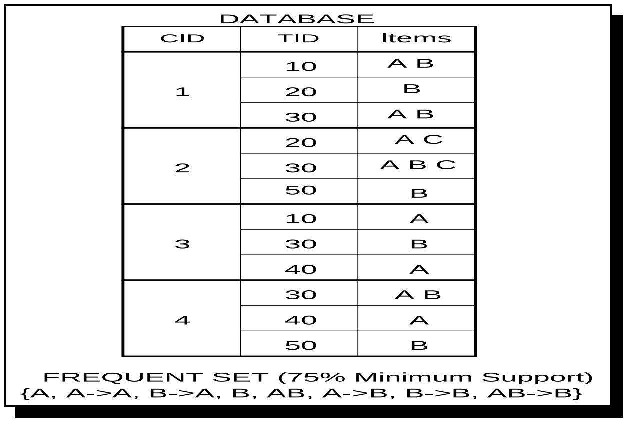
In this paper we present pSPADE, a parallel algorithm for discovering the set of all frequent sequences, targeting shared-memory systems, the first such study. pSPADE has been designed such that it has very good locality and has virtually no false sharing. Further, pSPADE is an asynchronous algorithm, in that it requires no synchronization among processors, except when a load imbalance is detected. We carefully consider several design alternatives in terms of data and task parallelism, and in terms of the load balancing strategy used, before adopting the best approach in pSPADE. An extensive set of experiments is performed on the SGI Origin 2000 machine. pSPADE delivers good speedup and scales linearly in the database size.

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

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). A *sequence* is an ordered list of itemsets. 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 $a_j \subseteq b_{i_j}$ for all a_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 $\beta \not\preceq \alpha$. A sequence is *maximal* if it is not a subsequence of any other sequence.


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}

Figure 1: Original Database

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 The Serial SPADE Algorithm

Several sequence mining algorithms have been proposed in recent years [Srikant and Agrawal, 1996, Mannila *et al.*, 1997, Oates *et al.*, 1997]. GSP [Srikant and Agrawal, 1996] is one of the best known serial algorithms. Recently, SPADE [Zaki, 1998] 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. In this section we describe SPADE [Zaki, 1998], 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 $\sigma(\alpha, \mathcal{D})$, 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 2 A) shows the lattice of frequent sequences for our example database.

Support Counting Most of the current sequence mining algorithms [Srikant and Agrawal, 1996] 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. The vertical idlist format, called *binary association table*, has also been used in [Holsheimer *et al.*, 1996] for parallel data mining. Figure 2 B) shows the idlists for all the 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 tells us whether the new sequence is frequent or not. Figure 2 shows this process pictorially. It shows the initial vertical database with the idlist for each item. There are two kinds of intersections: *temporal* and *equality*. For example, Figure 2 shows the idlist for $A \rightarrow B$ obtained by performing a temporal intersection on the idlists of A and B, i.e., $\mathcal{L}(A \rightarrow B) = \mathcal{L}(A) \cap_t \mathcal{L}(B)$. This is done by looking if, within the same *cid*, A occurs before B, and listing all such occurrences. On the other hand the idlist for $AB \rightarrow B$ is obtained by an equality intersection, i.e., $\mathcal{L}(AB \rightarrow B) = \mathcal{L}(A \rightarrow B) \cap_e \mathcal{L}(B \rightarrow B)$. Here we check to see if the two subsequences occur within the same *cid* at the same time. Additional details can be found in [Zaki, 1998].

To use only a limited amount of main-memory SPADE breaks up the sequence search space into small, independent, manageable chunks which can be processed in memory. This is accomplished via suffix-based partition. We say that two k length sequences are in the same equivalence class or *partition* if they share a common $k-1$ length suffix. The partitions, such as $\{[A], [B]\}$, based on length 1 suffixes are called *parent partitions*. Each parent partition 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 $Y \rightarrow X$, and $Z \rightarrow X$. The possible frequent sequences at the next step are $Y \rightarrow Z \rightarrow X$, $Z \rightarrow Y \rightarrow X$, and $(YZ)\ rightarrowX$. No other item Q can lead to a frequent sequence with the suffix X, unless (QX) or $Q \rightarrow X$ is also in $[X]$.

SPADE recursively decomposes the sequences at each new level into even smaller independent classes, which produces a computation tree of independent classes as shown in Figure 2B). This computation tree is processed in a breadth-first manner within each parent class. Figure 3 shows the pseudo-code for SPADE; we refer the reader to [Zaki, 1998] for more details.

4 The Parallel pSPADE Algorithm

While parallel association mining has attracted wide attention [Agrawal and Shafer, 1996, Zaki *et al.*, 1997],

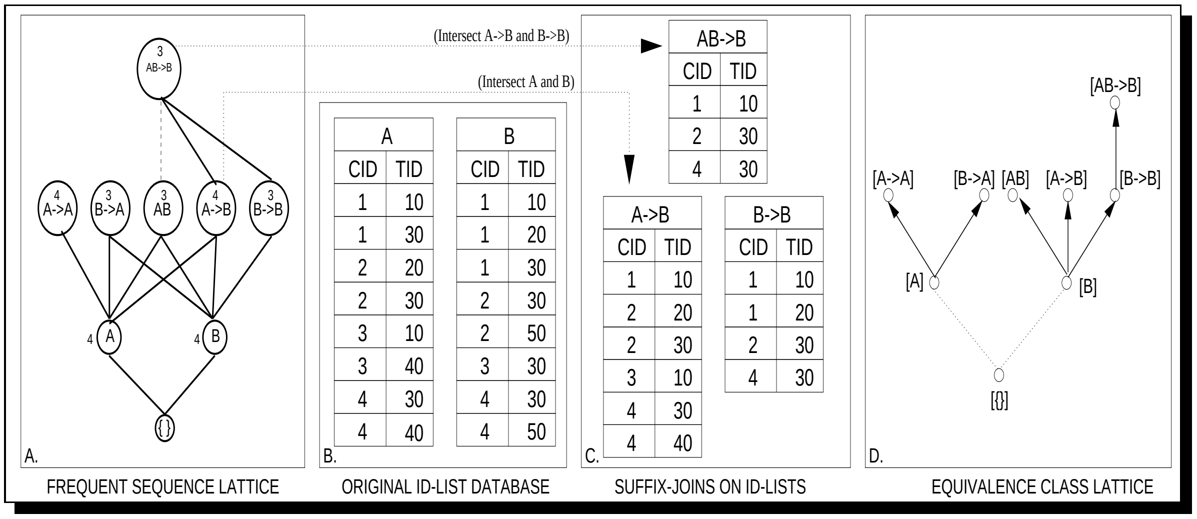
```
SPADE ($min\_sup, \mathcal{D}$):
  $\mathcal{C}$ = { parent classes $C_i = [X_i]$};
  for each $C_i \in \mathcal{C}$ do Enumerate-Frequent-Seq($C_i$);

//$PrevL$ 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 atoms $A_i \in S$ do
    for all atoms $A_j \in S$, with $j > i$ 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 $CList$;
```

Figure 3: SPADE pseudo-code

there has been relatively less work on parallel mining of sequential patterns. Three distributed-memory parallel algorithms based on GSP were presented in [Shintani and Kitsuregawa, 1998]. pSPADE is the first algorithm for shared-memory systems.

pSPADE is best understood by visualizing the computation as a dynamically expanding irregular tree of independent suffix-based classes, as shown in Figure 4. This tree represents the search space for the algorithm. There are three independent suffix-based parent 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 and a *task parallel* approach. 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



A	
CID	TID
1	10
1	30
2	20
2	30
3	10
3	40
4	30
4	40

B	
CID	TID
1	10
1	20
1	30
2	30
2	50
3	30
4	30
4	50

AB->B	
CID	TID
1	10
2	30
4	30

A->B	
CID	TID
1	10
2	20
2	30
3	10
4	30
4	40

B->B	
CID	TID
1	10
1	20
2	30
4	30

Figure 2: SPADE: Lattices and Joins

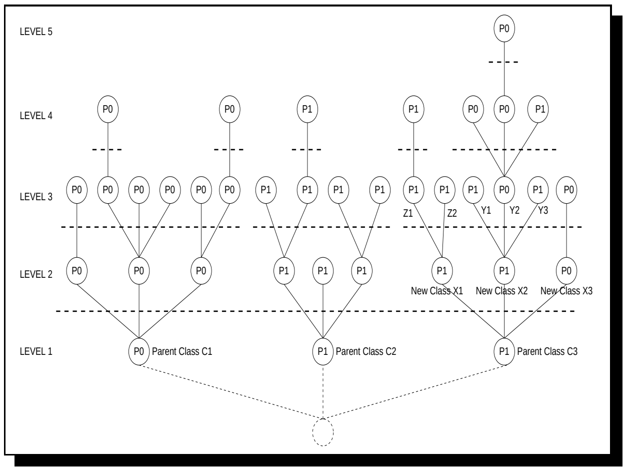


Figure 4: Dynamic, Irregular Computation Tree of Classes

based on inter-class parallelism.

4.0.1 Data Parallelism

For sequence mining, data parallelism can come in two flavors. The first case corresponds to *idlist parallelism*, in which we physically 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 *join parallelism*, where each processor picks a sequence and performs intersections with the other sequences 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 each 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 or added to the new class. This scheme suffers from massive synchronization overheads. As we shall see in Section 5 for some values of minimum support we performed around 0.6 million intersections. This scheme will require as many barrier synchronizations. The other method uses a level-wise approach. In other words, at each new level of the computation tree, each processor processes all the classes at that level, performing intersections for each candidate, but only over its local database portion. 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.

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 5). 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 extremely high. We were 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.

Join Parallelism In join parallelism each processor performs intersections for different sequences within the same class. Once the current class has been processed, the processors must synchronize before moving on to the next class. 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 single candidate idlist parallelism, but still much worse than the level-wise idlist parallelism, since there can be many classes.

4.0.2 Task Parallelism

In task parallelism all processors have access to the entire 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.

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 4. We need to schedule the classes among the processors in a manner minimizing load imbalance. In our approach an entire class is scheduled on one processor. Load balancing is achieved by assigning a weight to each 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 $W1_i = \binom{|C_i|}{2}$ 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. We also studied the effect of other heuristics for assigning class weights, such as $W2_i = \sum_j |\mathcal{L}(A_j)|$ 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., $W3_i = \binom{|C_i|}{2} \cdot \sum_j |\mathcal{L}(A_j)|$. We did not observe any significant benefit of one weight function over the other, and decided to use $W1$.

Figure 5 shows the pseudo-code for the SLB algorithm. We schedule the classes on different processors based on the class weights. Once the 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 $W1$ to the class tree shown in Figure 4, we get $W1_1 = W1_2 = W1_3 = 3$. Using the greedy scheduling scheme on two processors, P_0 gets the classes C_1 and C_3, and P_1 gets the class C_2. 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 per class statically.

Inter-Class Dynamic Load Balancing (CDLB) To get better load balancing we 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 class to work on from the list of 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. Note that each class usually has a non-trivial amount of work, so that we don't have to worry too much 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 5. 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. CAS ensures that processors acquire separate classes.

If we apply CDLB to our example computation tree in Figure 4, 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 5 that CDLB can provide up to 40% improvement over SLB. We should reiterate that the processing of classes is still asynchronous. For both

```
**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**
    $P_j$ = *Proc-with-Min-Weight()*;
    $S_{P_j} = S_{P_j} \cup C_i$;
  **parallel for** all $C_i \in S_{P_j}$ **do**
    *Enumerate-Frequent-Seq*$(C_i)$;
```

```
**CDLB** $(min\_sup, \mathcal{D})$:
  $\mathcal{C} = \{$ parent classes $C_i = [X_i]\}$;
  *Sort-on-Weight*$(\mathcal{C})$;

  **shared int** classid=0;
  **parallel for** $(i = 0; i < |\mathcal{C}|; i++)$
    **if** (*compare_and_swap* $(classid, i, i+1)$)
      *Enumerate-Frequent-Seq*$(C_i)$;
```

Figure 5: The SLB (Static Load Balancing) and CDLB (Dynamic Load Balancing) Algorithms

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, lets examine the behavior of CDLB. As long as there are more parent classes remaining, each processor acquires a new class and processes it completely. 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 smaller ones are at the end, it can still happen). We can fix this problem if we can 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 that the new level.

Figure 6 shows the pseudo-code for the pSPADE algorithm, which uses a recursive dynamic load balancing (RDLB) scheme. We start with the parent classes and insert them in the global class list, $GlobalQ$. A processor atomically acquires classes from this list until all parent classes have been taken. At that point the processor increments $FreeCnt$, and waits for more work. When a processor is processing the classes at some level, it periodically checks if there is any free processor. If so, it inserts the remaining classes at that level ($PrevL$) in $GlobalQ$, emptying $PrevL$ in the process, and sets $GlobalFlg$. This processor then quits the loop on line 16, and continues working on the new classes ($NewL$)

```
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** $(min\_sup, \mathcal{D})$:
4.   $GlobalQ = \mathcal{C} = \{$ parent classes $C_i = [X_i]\}$;
5.   *Sort-on-Weight*$(\mathcal{C})$;
6.   *Process-GlobalQ()*;
7.   $FreeCnt++$;
8.   **while** $(FreeCnt \neq P)$
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*$(C_i)$;
15.    $GlobalFlg = 0$;
**RDLB-Enumerate-Frequent-Seq**$(PrevL)$**:**
16.    **for** $(;\ PrevL \neq \emptyset;\ PrevL = PrevL.next())$
17.      **if** $(FreeCnt > 0)$ **then**
18.        *Add-to-GlobalQ*$(PrevL.next())$; $GlobalFlg = 1$;
19.      $NewL = NewL\ \cup$ *Get-New-Classes* $(PrevL.item())$;
20.    **if** $(NewL \neq \emptyset)$ **then** *RDLB-Enumerate-Frequent-Seq*$(NewL)$;
```

Figure 6: The pSPADE Algorithm (using RDLB)

generated before a free processor was detected. When a waiting processor sees that there is more work, it starts working on the classes in $GlobalQ$. When there is no more work, $FreeCnt$ equals the number of processors P, and the computation stops.

Let's illustrate the above algorithm by looking at the computation tree in Figure 4. 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 start 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

Dataset	C	T	S	I	D	Size	MinSup
C10T5S4I1.25D1M	10	5	4	1.25	1M	320MB	0.25%
C10T5S4I2.5D1M	10	5	4	2.5	1M	320MB	0.33%
C20T2.5S4I1.25D1M	20	2.5	4	1.25	1M	440MB	0.25%
C20T2.5S4I2.5D1M	20	2.5	4	2.5	1M	440MB	0.25%
C20T5S8I1.25D1M	20	5	8	1.25	1M	640MB	0.33%
C20T5S8I2D1M	20	5	8	2	1M	640MB	0.5%
C5T2.5S4I1.25DxM	5	2.5	4	1.25	1M-10M	110MB-1.1GB	0.25%-0.01%

Table 1: Synthetic Datasets

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. Figure 4 shows a possible execution sequence for the class C_3. 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. So far only inter-class parallelism has been exploited. 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.

5 Experimental Results

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 we stored on an attached 7GB disk, and all I/O is serial. Further, there were only 8 free processors available to us.

Synthetic Datasets We used the publicly available dataset generation code from the IBM Quest data mining project [IBM,]. 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. We set $N_S = 5000$, $N_I = 25000$ and $N = 10000$. Table 1 shows the datasets with their parameter settings.

Parallel Performance In [Zaki, 1998] SPADE was shown to outperform GSP, thus we chose not to parallelize GSP for comparison against pSPADE. As we mentioned in Section 4.0.1, the level-wise idlist data parallel algorithm performs very poorly, resulting in a speed-down with more processors. Results on 1, 2, and 4 processors shown in Figure 7 confirm this.

We now look at the parallel performance of pSPADE. Figure 8 shows the total execution time and the speedup charts for each database on the minimum support values shown in Table 1. We obtain near perfect speedup for 2 processors, ranging as high as 1.92. On 4 processors, we obtained a maximum of 3.5, and on 8 processors the maximum was 5.4. As these charts indicate, pSPADE achieves good speedup performance. Finally, we study the effect of dynamic load balancing on the parallel performance. Figure 7 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.

Scaleup Results Figure 9 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).

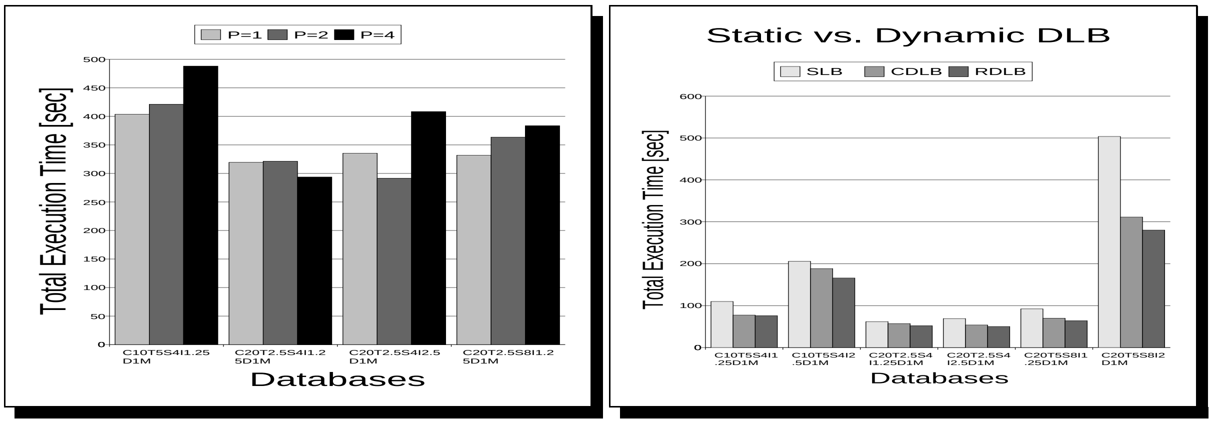


Figure 7: A) Level-Wise Idlist Data Parallelism, B) Effect of Load Balancing

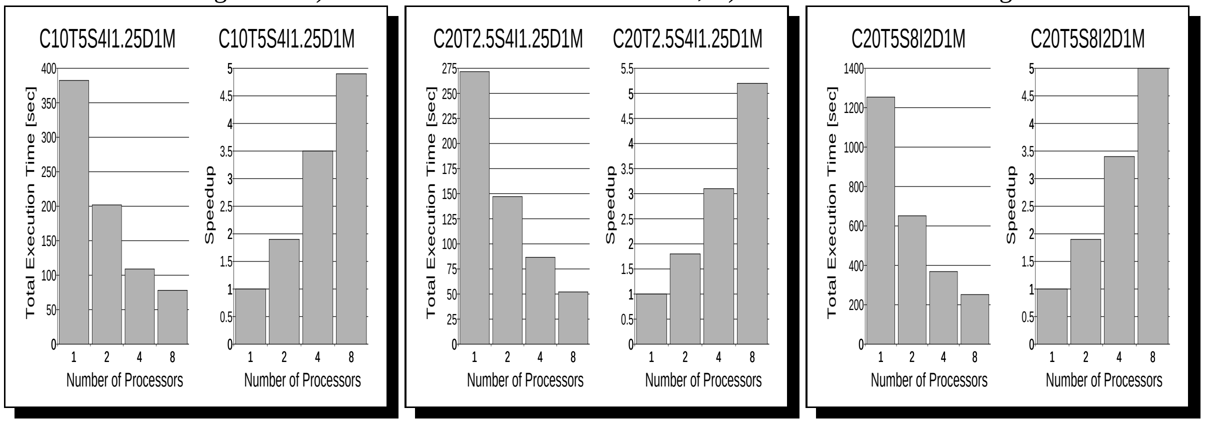


Figure 8: pSPADE Parallel Performance

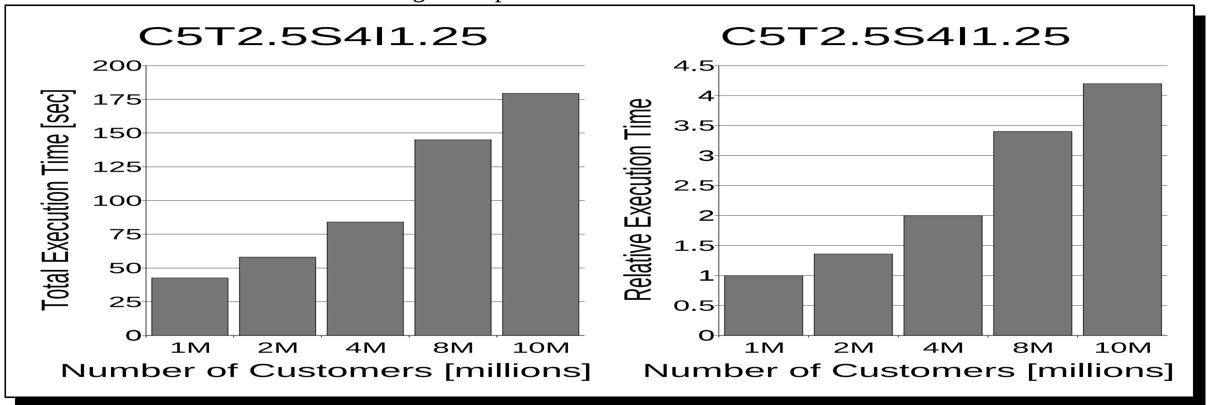


Figure 9: Sizeup

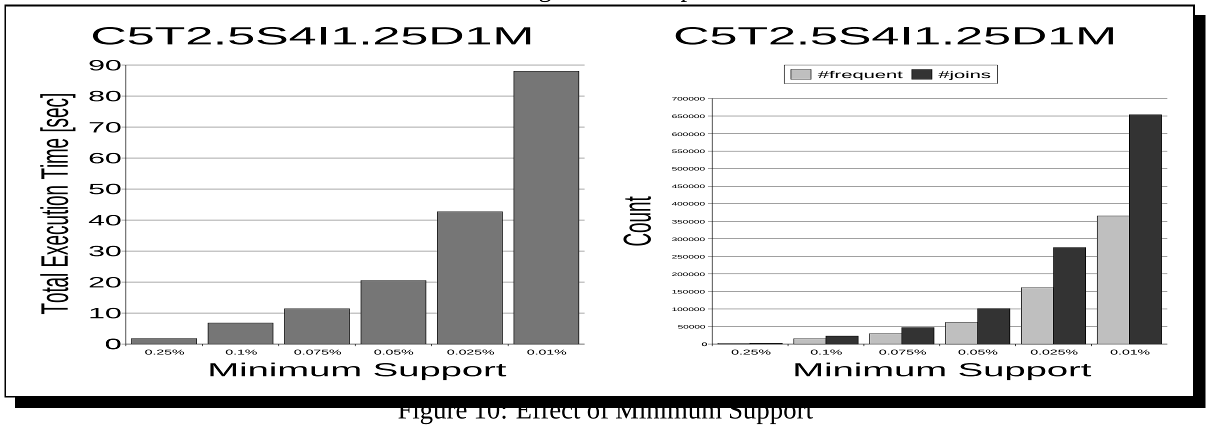


Figure 10: Effect of Minimum Support

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 sub-linear scaleup.

Finally, we study the effect of changing minimum support on the parallel performance, shown in Figure 10. 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 10 also shows the number of frequent sequences discovered and the number of joins performed (candidate sequences) at the different minimum support levels. 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/candidate sequences are not linear with respect to the minimum support.

6 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. It also has no false sharing. Finally, the scheme minimizes load imbalance by exploiting both inter-class and intra-class parallelism. Experiments conducted on the SGI Origin CC-NUMA shared memory system show that pSPADE has good speedup and scaleup properties.

References

[Agrawal and Shafer, 1996] R. Agrawal and J. Shafer. Parallel mining of association rules. *IEEE Trans. on Knowledge and Data Engg.*, 8(6):962–969, December 1996.

[Holsheimer *et al.*, 1996] M. Holsheimer, M. L. Kersten, and A. Siebes. Data surveyor: Searching the nuggets in parallel. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*. AAAI Press, Menlo Park, CA, 1996.

[IBM,] IBM. *http://www.almaden.ibm.com/cs/quest/-syndata.html*. Quest Data Mining Project, IBM Almaden Research Center, San Jose, CA 95120.

[Mannila *et al.*, 1997] H. Mannila, H. Toivonen, and I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery: An International Journal*, 1(3):259–289, 1997.

[Oates *et al.*, 1997] T. Oates, M. D. Schmill, D. Jensen, and P. R. Cohen. A family of algorithms for finding temporal structure in data. In *6th Intl. Workshop on AI and Statistics*, March 1997.

[Shintani and Kitsuregawa, 1998] T. Shintani and M. Kitsuregawa. Mining algorithms for sequential patterns in parallel: Hash based approach. In *Pacific-Asia Conf. on Knowledge Discovery and Data Mining*, April 1998.

[Srikant and Agrawal, 1996] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *5th Intl. Conf. Extending Database Technology*, March 1996.

[Zaki *et al.*, 1997] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. Parallel algorithms for fast discovery of association rules. *Data Mining and Knowledge Discovery: An International Journal*, 1(4):343-373, December 1997.

[Zaki, 1998] M. J. Zaki. Efficient enumeration of frequent sequences. In *7th Intl. Conf. on Information and Knowledge Management*, November 1998.