Knowledge and Information Systems (2001) 3: 1–29
© 2001 Springer-Verlag London Ltd.

Knowledge and Information Systems

Parallel Data Mining for Association Rules on Shared-Memory Systems

S. Parthasarathy[1], M. J. Zaki[2], M. Ogihara[3], W. Li[4]

[1]Department of Computer and Information Sciences, Ohio State University, Columbus, OH, USA
[2]Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY, USA
[3]Department of Computer Science, University of Rochester, Rochester, NY, USA
[4]Intel Corporation, Santa Clara, CA, USA

Abstract. In this paper we present a new parallel algorithm for data mining of association rules on shared-memory multiprocessors. We study the degree of parallelism, synchronization, and data locality issues, and present optimizations for fast frequency computation. Experiments show that a significant improvement of performance is achieved using our proposed optimizations. We also achieved good speed-up for the parallel algorithm.

A lot of data-mining tasks (e.g. association rules, sequential patterns) use complex pointer-based data structures (e.g. hash trees) that typically suffer from suboptimal *data locality*. In the multiprocessor case shared access to these data structures may also result in *false sharing*. For these tasks it is commonly observed that the recursive data structure is built once and accessed multiple times during each iteration. Furthermore, the access patterns after the build phase are highly ordered. In such cases locality and false sharing sensitive memory placement of these structures can enhance performance significantly. We evaluate a set of placement policies for parallel association discovery, and show that simple placement schemes can improve execution time by more than a factor of two. More complex schemes yield additional gains.

Keywords: Association rules; Improving locality; Memory placement; Parallel data mining; Reducing false sharing

1. Introduction

Discovery of association rules is an important problem in database mining. The prototypical application is the analysis of sales or *basket* data (Agrawal et al, 1996). Basket data consists of items bought by a customer along with the

Received 24 May 1999
Revised 20 Jun 2000
Accepted 6 Jul 2000

transaction identifier. Besides the retail sales example, association rules have been shown to be useful in domains such as decision support, telecommunications alarm diagnosis and prediction, university enrollments, etc.

Due to the huge size of data and amount of computation involved in data mining, parallel computing is a crucial component for any successful large-scale data-mining application. Past research on parallel association mining (Park et al, 1995b; Agrawal and Shafer, 1996; Cheung et al, 1996b; Han et al, 1997; Zaki et al, 1997e) has been focused on distributed-memory (also called shared-nothing) parallel machines. In such a machine, each processor has private memory and local disks, and communicates with other processors only via passing messages. Parallel distributed-memory machines are essential for scalable massive parallelism. However, shared-memory multiprocessor systems (SMPs), often called shared-everything systems, are also 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 even parallel distributed-memory machines are increasingly being designed to be SMP nodes. A shared-memory system offers a single memory address space that all processors can access. Processors communicate through shared variables in memory and are capable of accessing any memory location. Synchronization is used to coordinate processes. Any processor can also access any disk attached to the system. The programming architecture on these machines is quite different from that on distributed-memory machines, which warrants the development of new algorithms for achieving good performance.

Modern processor speeds are improving by 50–100% every year, while memory access times are improving by only 7% every year (Hennessey and Patterson, 1995). Memory latency (due to the gap between processor and memory subsystems) is therefore becoming an increasingly important performance bottleneck in modern multiprocessors. While cache hierarchies alleviate this problem to an extent, *data locality* optimization (i.e., having as much as possible of the data local to the processor cache) is crucial for improving performance of data-intensive applications like data mining. Furthermore, in shared-memory (SMP) systems the local processor caches have to be kept coherent with one another. A key to efficiency is to minimize the cost of maintaining coherency. The elimination of *false sharing* (i.e., the problem where coherence is maintained even when memory locations are not shared) is an important step towards achieving this objective.

This paper presents a new parallel algorithm for mining association rules on shared-memory multiprocessor systems. We present novel techniques for load balancing the computation in the enumeration of frequent associations. We additionally present new optimizations for balancing the hash tree data structure to enable fast counting. The recursive data structures found in association mining (such as hash trees, lists, etc.) typically exhibit poor locality, and shared access to these structures may also introduce false sharing. Unfortunately traditional locality optimizations found in the literature (Carr et al, 1994; Anderson and Lam, 1993) cannot be used directly for such structures, since they can be applied only to array-based programs. The arbitrary allocation of memory in data-mining applications also makes detecting and eliminating false sharing a difficult proposition. In this paper we also describe techniques for improving locality and reducing false sharing. We propose a set of policies for controlling the allocation and memory placement of dynamic data structures. The schemes are based on a detailed understanding of the access patterns of the program, and memory is allocated in such a way that data likely to be accessed together is grouped together (memory placement) while minimizing false sharing.

We experimentally evaluate the performance of our new algorithm and optimizations. We achieve excellent speed-up of up to eight on 12 processors. We show that the proposed optimizations for load balancing, hash tree balancing, and fast counting are extremely effective, with up to 40% improvements over the base case. For the locality-enhancing and false-sharing minimizing techniques, we show that simple placement schemes can be quite effective, and for the datasets we looked at improve the execution time by a factor of two. More complex schemes yield additional gains. These results are directly applicable to other mining tasks like quantitative associations (Srikant and Agrawal, 1996), multi-level (taxonomies) associations (Srikant and Agrawal, 1995), and sequential patterns (Agrawal and Srikant, 1995), which also use hash tree-based structures.

The rest of the paper is organized as follows. We describe the sequential association algorithm in Section 2, and the parallel algorithm in Section 3. The optimizations for balancing the hash tree, and fast subset checking, are presented in Section 4. Section 5 describes the custom placement policies for improving locality and reducing false sharing. We next present our experimental results in Section 6. Relevant related work is discussed in Section 7, and we present our conclusions in Section 8.

2. Sequential Association Mining

The problem of mining associations over basket data was introduced in Agrawal et al (1993). It can be formally stated as: Let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of m distinct attributes, also called *items*. Each transaction T in the database $\mathcal{D}$ of transactions has a unique identifier, and *contains* a set of items, called an *itemset*, such that $T \subseteq \mathcal{I}$, i.e. each transaction is of the form $< \mathrm{TID}, i_1, i_2, \ldots, i_k >$. An itemset with k items is called a k-*itemset*. A subset of length k is called a k-*subset*. An itemset is said to have a *support* s if s% of the transactions in $\mathcal{D}$ contain the itemset. An *association rule* is an expression $A \Rightarrow B$, where itemsets $A, B \subset \mathcal{I}$, and $A \cap B = \emptyset$. The *confidence* of the association rule, given as $support(A \cup B)/support(A)$, is simply the conditional probability that a transaction contains B, given that it contains A. The data-mining task for association rules can be broken into two steps. The first step consists of finding all *frequent* itemsets, i.e., itemsets that occur in the database with a certain user-specified frequency, called *minimum support*. The second step consists of forming implication rules among the frequent itemsets (Agrawal and Srikant, 1994). The second step is relatively straightforward. Once the support of frequent itemsets is known, rules of the form $X - Y \Rightarrow Y$ (where $Y \subset X$) are generated for all frequent itemsets X, provided the rules meet the desired confidence. On the other hand the problem of identifying all frequent itemsets is hard. Given m items, there are potentially 2^m frequent itemsets. However, only a small fraction of the whole space of itemsets is frequent. Discovering the frequent itemsets requires a lot of computation power, memory and I/O, which can only be provided by parallel computers.

2.1. Sequential Association Algorithm

Our parallel shared-memory algorithm is built on top of the sequential *Apriori* association mining algorithm proposed in Agrawal et al (1996). During iteration

```
$\mathscr{F}_1 = \{$frequent 1-itemsets $\}$;
for $(k = 2; \mathscr{F}_{k-1} \neq \emptyset; k++)$
    $C_k =$ Set of New Candidates;
    for all transactions $t \in \mathscr{D}$
        for all $k$-subsets $s$ of $t$
            if $(s \in C_k)$ $s.count++$;
    $\mathscr{F}_k = \{c \in C_k | c.count \geqslant min\_sup\}$;
Set of all frequent itemsets $= \bigcup_k \mathscr{F}_k$;
```

Fig. 1. Sequential association mining.

k of the algorithm a set of candidate k-itemsets is generated. The database is then scanned and the support for each candidate is found. Only the frequent k-itemsets are retained for future iterations, and are used to generate a candidate set for the next iteration. A pruning step eliminates any candidate which has an infrequent subset. This iterative process is repeated for $k = 1, 2, 3 \ldots$, until there are no more frequent k-itemsets to be found. The general structure of the algorithm is given in Fig. 1. In the figure F_k denotes the set of frequent k-itemsets, and C_k the set of candidate k-itemsets. There are two main steps in the algorithm: candidate generation and support counting.

2.1.1. Candidate Itemset Generation

In candidate itemsets generation, the candidates C_k for the k-th pass are generated by joining F_{k-1} with itself, which can be expressed as

$$C_k = \{x \mid x[1:k-2] = A[1:k-2] = B[1:k-2],\ x[k-1] = A[k-1],\\ x[k] = B[k-1],\ A[k-1] < B[k-1],\ \text{where } A, B \in F_{k-1}\}$$

where $x[a:b]$ denotes items at index a through b in itemset x. Before inserting x into C_k, we test whether all $(k-1)$-subsets of x are frequent. If there is at least one subset that is not frequent the candidate can be pruned.

Hash tree data structure: The candidates are stored in a hash tree to facilitate fast support counting. An internal node of the hash tree at depth d contains a hash table whose cells point to nodes at depth $d+1$. The size of the hash table, also called the *fan-out*, is denoted as $\mathscr{H}$. All the itemsets are stored in the leaves in a sorted linked list. To insert an itemset in C_k, we start at the root, and at depth d we hash on the d-th item in the itemset until we reach a leaf. The itemset is then inserted in the linked list of that leaf. If the number of itemsets in that leaf exceeds a *threshold* value, that node is converted into an internal node. We would generally like the fan-out to be large, and the threshold to be small, to facilitate fast support counting. The maximum depth of the tree in iteration k is k. Figure 2 shows a hash tree containing candidate 3-itemsets, and Fig. 3 shows the structure of the internal and leaf nodes in more detail. The different components of the hash tree are as follows: hash tree node (**HTN**), hash table (**HTNP**), itemset list header (**ILH**), list node (**LN**), and the itemsets (**Itemset**). It can be seen that an internal node has a hash table pointing to nodes at the next level, and an empty itemset list, while a leaf node has a list of itemsets.

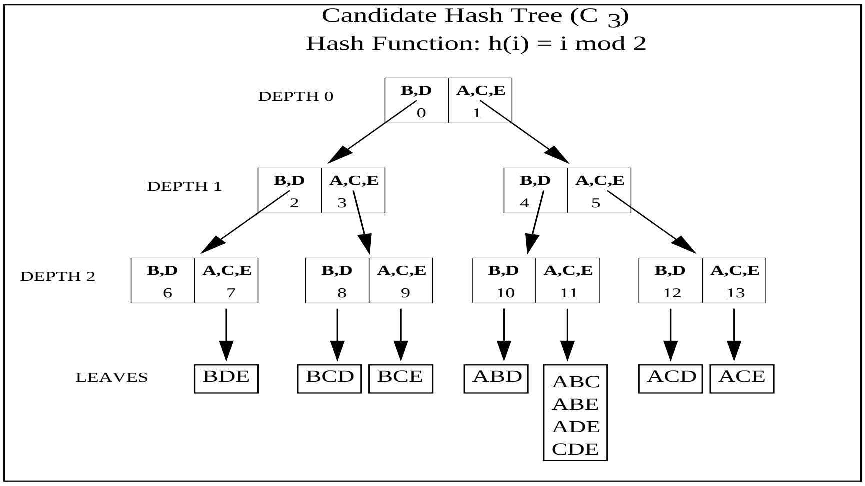

Fig. 2. Candidate hash tree.

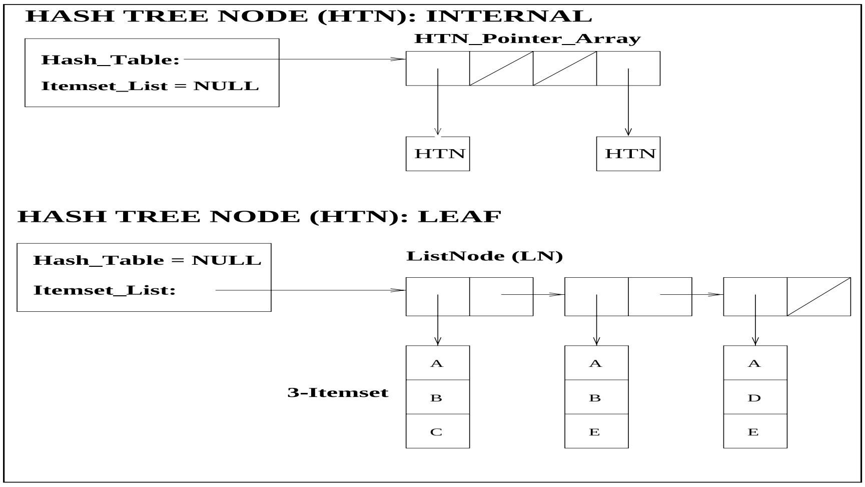

Fig. 3. Structure of internal and leaf nodes.

2.1.2. Support Counting

To count the support of candidate k-itemsets, for each transaction T in the database, we conceptually form all k-subsets of T in lexicographical order. For each subset we then traverse the hash tree and look for a matching candidate, and update its count.

Hash tree traversal: The search starts at the root by hashing on successive items of the transaction. At the root we hash on all transaction items from 0 through $(n - k + 1)$, where n is the transaction length and k the current iteration. If we reach depth d by hashing on item i, then we hash on the remaining items i through $(n - k + 1) + d$ at that level. This is done recursively, until we reach a leaf. At this point we traverse the linked list of itemsets and increment the count of all itemsets that are contained in the transaction (note: this is the reason for having a small leaf threshold value).

2.1.3. Frequent Itemset Example

Once the support for the candidates has been gathered, we traverse the hash tree in depth first order, and all candidates that have the minimum support are inserted in F_k, the set of frequent k-itemsets. This set is then used for candidate generation in the next iteration.

For a simple example demonstrating the different steps, consider an example database, $\mathscr{D} = \{T_1 = (1,4,5), T_2 = (1,2), T_3 = (3,4,5), T_4 = (1,2,4,5)\}$, where T_i denotes transaction i. Let the minimum support value be 2. Running through the iterations, we obtain

$$\begin{aligned}
F_1 &= \{(1),(2),(4),(5)\} \\
C_2 &= \{(1,2),(1,4),(1,5),(2,4),(2,5),(4,5)\} \\
F_2 &= \{(1,2),(1,4),(1,5),(4,5)\} \\
C_3 &= \{(1,4,5)\} \\
F_3 &= \{(1,4,5)\}
\end{aligned}$$

Note that while forming C_3 by joining F_2 with itself, we get three potential candidates, $(1,2,4)$, $(1,2,5)$, and $(1,4,5)$. However only $(1,4,5)$ is a true candidate. The first two are eliminated in the pruning step, since they have a 2-subset which is not frequent, namely the 2-subsets $(2,4)$ and $(2,5)$, respectively.

3. Parallel Association Mining on SMP Systems

In this section we present the design issues in parallel data mining for association rules on shared memory architectures. We separately look at the two main steps: candidate generation, and support counting.

3.1. Candidate Itemsets Generation

3.1.1. Optimized Join and Pruning

Recall that in iteration k, C_k is generated by joining F_{k-1} with itself. The naive way of doing the join is to look at all $\binom{|F_{k-1}|}{2}$ combinations. However, since F_{k-1} is lexicographically sorted, we can partition the itemsets in F_{k-1} into equivalence classes $S_0, \ldots, S_n$, based on their common $k-2$ prefixes (class identifier). k-itemsets are formed only from items within a class by taking all $\binom{|S_i|}{2}$ item pairs and prefixing them with the class identifier. In general we have to consider $\sum_{i=0}^{n} \binom{|S_i|}{2}$ combinations instead of $\binom{|F_{k-1}|}{2}$ combinations.

While pruning a candidate we have to check if all k of its $(k-1)$-subsets are frequent. Since the candidate is formed by an item pair from the same class, we need only check for the remaining $k-2$ subsets. Furthermore, assuming all S_i are lexicographically sorted, these $k-2$ subsets must come from classes greater than the current class. Thus, to generate a candidate, there must be at least $k-2$

equivalence classes after a given class. In other words we need consider only the first $n-(k-2)$ equivalence classes.

Adaptive hash table size ($\mathscr{H}$): Having equivalence classes also allows us to accurately adapt the hash table size $\mathscr{H}$ for each iteration. For iteration k, and for a given *threshold* value $\mathscr{T}$, i.e., the maximum number of k-itemsets per leaf, the total k-itemsets that can be inserted into the tree is given by the expression $\mathscr{T}\mathscr{H}^k$. Since we can insert up to $\sum_{i=0}^{n}\binom{|S_i|}{2}$ itemsets, we get the expression $\mathscr{T}\mathscr{H}^k \geqslant \sum_{i=0}^{n}\binom{|S_i|}{2}$. This can be solved to obtain

$$\mathscr{H} \geqslant \left(\frac{\sum_{i=0}^{n}\binom{|S_i|}{2}}{\mathscr{T}}\right)^{1/k}$$

3.1.2. Computation Balancing

Let the number of processors $\mathscr{P}=3$, $k=2$, and $F_1=\{0,1,2,3,4,5,6,7,8,9\}$. There is only one resulting equivalence class since the $k-2$ $(=0)$ length common prefix is null. The number of 2-itemsets generated by an itemset, called the *work load* due to itemset i, is given as $w_i = n-i-1$, for $i=0,\ldots,n-1$. For example, itemset 0 contributes nine 2-itemsets $\{01,02,03,04,05,06,07,08,09\}$. There are different ways of partitioning this class among $\mathscr{P}$ processors.

Block partitioning: A simple block partition generates the assignment $\mathscr{A}_0 = \{0,1,2\}$, $\mathscr{A}_1=\{3,4,5\}$ and $\mathscr{A}_2=\{6,7,8,9\}$, where $\mathscr{A}_p$ denotes the itemsets assigned to processor p. The resulting workload per processor is: $\mathscr{W}_0 = 9+8+7 = 24$, $\mathscr{W}_1 = 6+5+4 = 15$ and $\mathscr{W}_2 = 3+2+1 = 6$, where $\mathscr{W}_p = \sum_{i\in\mathscr{A}_p} w_i$. We can clearly see that this method suffers from a load imbalance problem.

Interleaved partitioning: A better way is to do an interleaved partition, which results in the assignment $\mathscr{A}_0 = 0,3,6,9$, $\mathscr{A}_1 = 1,4,7$ and $\mathscr{A}_2 = 2,5,8$. The workload is now given as $\mathscr{W}_0 = 9+6+3 = 18$, $\mathscr{W}_1 = 8+5+2 = 15$ and $\mathscr{W}_2 = 7+4+1 = 12$. The load imbalance is much smaller; however, it is still present.

Bitonic partitioning (single equivalence class): In Cierniak et al (1997) we propose a new partitioning scheme, called *bitonic partitioning*, for load balancing that can be applied to the problem here as well. This scheme is based on the observation that the sum of the workload due to itemsets i and $(2\mathscr{P}-i-1)$ is a constant:

$$w_i + w_{2\mathscr{P}-i-1} = n-i-1+(n-(2\mathscr{P}-i-1)-1) = 2n-2\mathscr{P}-1$$

We can therefore assign itemsets i and $(2P-i-1)$ as one *unit* with uniform work $(2n-2\mathscr{P}-1)$. If $n \bmod 2\mathscr{P} = 0$ then perfect balancing results. The case $n \bmod 2\mathscr{P} \neq 0$ is handled as described in Cierniak et al (1997).

The final assignment is given as $\mathscr{A}_0 = \{0,5,6\}$,$\mathscr{A}_1 = \{1,4,7\}$, and $\mathscr{A}_2 = \{2,3,8,9\}$, with corresponding workload given as $\mathscr{W}_0 = 9+4+3 = 16$, $\mathscr{W}_1 = 8+5+2 = 15$ and $\mathscr{W}_2 = 7+6+1 = 14$. This partition scheme is better than the interleaved scheme and results in almost no imbalance. Figure 4 illustrates the difference between the interleaved and bitonic partitioning schemes.

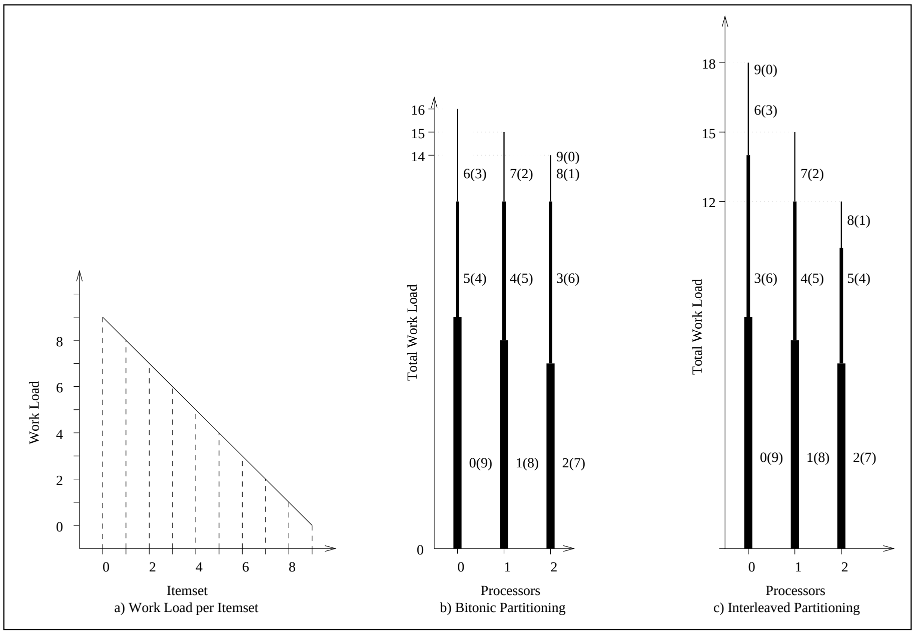

Fig. 4. Bitonic partitioning.

Bitonic partitioning (multiple equivalence classes): Above we presented the simple case of C_1, where we only had a single equivalence class. In general we may have multiple equivalence classes. Observe that the bitonic scheme presented above is a greedy algorithm, i.e., we sort all the w_i (the workload due to itemset i), extract the itemset with maximum w_i, and assign it to processor 0. Each time we extract the maximum of the remaining itemsets and assign it to the least loaded processor. This greedy strategy generalizes to the multiple equivalence class as well (Zaki et al, 1996), the major difference being workloads in different classes may not be distinct.

3.1.3. *Adaptive Parallelism*

Let n be the total number of items in the database. Then there are potentially $\binom{n}{k}$ frequent k-itemsets that we would have to count during iteration k. However, in practice the number is usually much smaller, as is indicated by our experimental results. We found that support counting dominated the execution time to the tune of around 85% of the total computation time for the databases we consider in Section 6. On the other hand, for iterations with a large number of k itemsets there was sufficient work in the candidate generation phase. This suggests a need for some form of dynamic or adaptive parallelization based on the number of frequent k-itemsets. If there are not a sufficient number of frequent itemsets, then it is better not to parallelize the candidate generation.

3.1.4. Parallel Hash Tree Formation

We could choose to build the candidate hash tree in parallel, or we could let the candidates be temporarily inserted in local lists (or hash trees). This would have to be followed by a step to construct the global hash tree.

In our implementation we build the tree in parallel. We associate a lock with each leaf node in the hash tree. When processor i wants to insert a candidate itemset into the hash tree it starts at the root node and hashes on successive items in the itemsets until it reaches a leaf node. At this point it acquires a lock on this leaf node for mutual exclusion while inserting the itemset. However, if we exceed the *threshold* of the leaf, we convert the leaf into an internal node (with the lock still set). This implies that we also have to provide a lock for all the internal nodes, and the processors will have to check if any node is acquired along its downward path from the root. With this locking mechanism, each process can insert the itemsets in different parts of the hash tree in parallel.

3.2. Support Counting

For this phase, we could either split the database logically among the processors with a common hash tree, or split the hash tree with each processor traversing the entire database. We will look at each case below.

3.2.1. Partitioned vs. Common Candidate Hash Tree

One approach in parallelizing the support counting step is to split the hash tree among the processors. The decisions for computation balancing directly influence the effectiveness of this approach, since each processor should ideally have the same number of itemsets in its local portion of the hash tree. Another approach is to keep a single common hash tree among all the processors.

3.2.2. Partitioned vs. Common Database

We could either choose to logically partition the database among the processors, or each processor can choose to traverse the entire database for incrementing the candidate support counts.

Balanced database partitioning: In our implementation we partition the database in a blocked fashion among all the processors. However, this strategy may not result in balanced work per processor. This is because the workload is a function of the length of the transactions. If l_t is the length of the transaction t, then during iteration k of the algorithm we have to test whether all the $\binom{l_t}{k}$ subsets of the transaction are contained in C_k. Clearly the complexity of the workload for a transaction is given as $\mathcal{O}(\min(l_t{}^k, l_t{}^{l_t-k}))$, i.e., it is polynomial in the transaction length. This also implies that a static partitioning will not work. However, we could devise static heuristics to approximate a balanced partition. For example, one static heuristic is to estimate the maximum number of iterations we expect, say T. We could then partition the database based on the mean estimated workload for each transaction over all iterations, given as $(\sum_{k=1}^{T} \binom{l_t}{k})/T$. Another approach is to re-partition the database in each iteration. In this case it is important to respect the locality of the partition by moving transactions only when it is absolutely necessary. We plan to investigate different partitioning schemes as part of future work.

3.3. Parallel Data Mining: Algorithms

Based on the discussion in the previous section, we consider the following algorithms for mining association rules in parallel:

- *Common candidate partitioned database (CCPD):* This algorithm uses a common candidate hash tree across all processors, while the database is logically split among them. The hash tree is built in parallel (see Section 3.1.4). Each processor then traverses its local database and counts the support for each itemset. Finally, the master process selects the frequent itemsets.
- *Partitioned Candidate Common Database (PCCD):* This has a partitioned candidate hash tree, but a common database. In this approach we construct a local candidate hash tree per processor. Each processor then traverses the entire database and counts support for itemsets only in its local tree. Finally the master process performs the reduction and selects the frequent itemsets for the next iteration.

Note that the common candidate common database (CCCD) approach results in duplicated work, while the partitioned candidate partitioned database (PCPD) approach is more or less equivalent to CCPD on shared-memory systems. For this reason we did not implement these parallelizations.

4. Optimizations: Tree Balancing and Fast Subset Checking

In this section we present some optimizations to the association rule algorithm. These optimizations are beneficial for both sequential and parallel implementation.

4.1. Hash Tree Balancing

Although the computation balancing approach results in balanced workload, it does not guarantee that the resulting hash tree is balanced.

Balancing C_2 (no pruning): We'll begin by a discussion of tree balancing for C_2, since there is no pruning step in this case. We can balance the hash tree by using the bitonic partitioning scheme described above. We simply replace P, the number of processors with the fan-out $\mathscr{H}$ for the hash table. We label the n frequent 1-itemsets from 0 to $n-1$ in lexicographical order, and use $P = \mathscr{H}$ to derive the assignments $\mathscr{A}_0, \ldots, \mathscr{A}_{\mathscr{H}-1}$ for each processor. Each $\mathscr{A}_i$ is treated as an equivalence class. The hash function is based on these equivalence classes, which is simply given as, $h(i) = \mathscr{A}_i$, for $i = 0, \ldots, \mathscr{H}$. The equivalence classes are implemented via an indirection vector of length n. For example, let $F_1 = \{A, D, E, G, K, M, N, S, T, Z\}$. We first label these as $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Assume that the fan-out $\mathscr{H} = 3$. We thus obtain the three equivalence classes $\mathscr{A}_0 = \{0, 5, 6\}$, $\mathscr{A}_1 = \{1, 4, 7\}$, and $\mathscr{A}_2 = \{2, 3, 8, 9\}$, and the indirection vector is shown in Table 1. Furthermore, this hash function is applied at all levels of the

Table 1. Indirection vector.

Label	0	1	2	3	4	5	6	7	8	9
Hash value	0	1	2	2	1	0	0	1	2	2

hash tree. Clearly, this scheme results in a balanced hash tree as compared to the simple $g(i) = i \bmod \mathscr{H}$ hash function (which corresponds to the interleaved partitioning scheme from Section 3.1.1).

Balancing $C_k (k > 2)$: Although items can be pruned for iteration $k \geqslant 3$, we use the same bitonic partitioning scheme for C_3 and beyond. Below we show that even in this general case bitonic hash function is very good as compared to the interleaved scheme. Theorem 1 below establishes an upper and lower bound on the number of itemsets per leaf for the bitonic scheme.

Theorem 1. Let $k \geqslant 1$ denote the iteration number, $\mathscr{I} = \{0, \ldots, d-1\}$ the set of items, $\mathscr{H}$ the fan-out of the hash table, $T = \{0, \ldots, \mathscr{H}-1\}$ the set of equivalence classes modulo $\mathscr{H}$, $\mathscr{T} = T^k$ the total number of leaves in C_k, and $\mathscr{G}$ the family of all size k ordered subsets of $\mathscr{I}$, i.e., the set of all k-itemsets that can be constructed from items in $\mathscr{I}$. Suppose $\frac{d}{2\mathscr{H}}$ is an integer and $\frac{d}{2\mathscr{H}}, \mathscr{H} \geqslant k$. Define the bitonic hash function $h : \mathscr{I} \to T$ by

$$h(i) = i \bmod \mathscr{H} \text{ if } 0 \leqslant (i \bmod 2\mathscr{H}) < \mathscr{H} \text{ and } 2\mathscr{H} - 1 - (i \bmod 2\mathscr{H}) \text{ otherwise,}$$

and the mapping $S : \mathscr{G} \to \mathscr{T}$ from k-itemsets to the leaves of C_k by $S(a_1, \ldots, a_k) = (h(a_1), \ldots, h(a_k))$. Then for every leaf $B = (b_1, \ldots, b_k) \in \mathscr{T}$, the ratio of the number of k-itemsets in the leaf ($\|S^{-1}(B)\|$) to the average number of itemsets per leaf ($\|\mathscr{G}\|/\|\mathscr{T}\|$) is bounded above and below by the expression

$$e^{-\frac{k^2}{d/\mathscr{H}}} \leqslant \frac{\|S^{-1}(B)\|}{\|\mathscr{G}\|/\|\mathscr{T}\|} \leqslant e^{\frac{k^2}{d/\mathscr{H}}}$$

A proof of the above theorem can be found in Zaki et al (1996). We also obtain the same lower and upper bound for the interleaved hash function. However, the two functions behave differently. Note that the average number of k-itemsets per leaf $\|\mathscr{G}\|/\|\mathscr{T}\|$ is $\binom{2w\mathscr{H}}{k}/\mathscr{H}^k \approx \frac{(2w)^k}{k!}$. Let $\alpha(w)$ denote this polynomial. We say that a leaf has a capacity close to the average if its capacity, which is a polynomial in w of degree at most k, is of the form $\frac{(2w)^k}{k!} + \beta(w)$, with $\beta(w)$ being a polynomial of degree at most $k-2$.

For the bitonic hash function, a leaf specified by the hash values $(a_1, \ldots, a_k)$ has capacity close to $\alpha(w)$ if and only if $a_i \neq a_{i+1}$ for all i, $1 \leqslant i \leqslant k-1$. Thus, there are $\mathscr{H}(\mathscr{H}-1)^{k-1}$ such leaves, and so, $(1 - \mathscr{H}^{-1})^{k-1}$ fraction of the leaves have capacity close to $\alpha(w)$. Note also that clearly, $(1 - \mathscr{H}^{-1})^{k-1}$ approaches 1.

On the other hand, for the interleaved hash function, a leaf specified by $(a_1, \ldots, a_k)$ has capacity close to $\alpha(w)$ if and only if $a_i \neq a_{i+1}$ for all i, and the number of i such that $a_i < a_{i+1}$ is equal to $(k-1)/2$. So, there is no such leaf if k is even. For odd $k \geqslant 3$, the ratio of the 'good' leaves decreases as k increases, achieving a maximum of $2/3$ when $k = 3$. Thus, at most $2/3$ of the leaves achieve the average.

From the above discussion it is clear that while both the simple and bitonic hash function have the same maximum and minimum bounds, the distribution of the number of itemsets per leaf is quite different. While a significant portion of the leaves are close to the average for the bitonic case, only a few are close in the simple hash function case.

4.2. Short-Circuited Subset Checking

Recall that while counting the support, once we reach a leaf node, we check whether all the itemsets in the leaf are contained in the transaction. This node is then marked as VISITED to avoid processing it more than once for the same transaction. A further optimization is to associate a VISITED flag with each node in the hash tree. We mark an internal node as VISITED the first time we touch it. This enables us to preempt the search as soon as possible. We would expect this optimization to be of greatest benefit when the transaction sizes are large. For example, if our transaction is $T = \{A, B, C, D, E\}$, $k = 3$, fan-out $= 2$, then all the 3-subsets of T are: {ABC, ABD, ABE, ACD, ACE, ADE, BCD, BCE, BDE, CDE}. Figure 2 shows the candidate hash tree C_3. We have to increment the support of every subset of T contained in C_3. We begin with the subset ABC, and hash to node 11 and process all the itemsets. In this downward path from the root we mark nodes 1, 4, and 11 as visited. We then process subset ADB, and mark node 10. Now consider the subset CDE. We see in this case that node 1 has already been marked, and we can preempt the processing at this very stage. This approach can, however, consume a lot of memory. For a given fan-out $\mathscr{H}$, for iteration k, we need additional memory of size $\mathscr{H}^k$ to store the flags. In the parallel implementation we have to keep a VISITED field for each processor, bringing the memory requirement to $\mathscr{P} \cdot \mathscr{H}^k$. This can still get very large, especially with increasing number of processors. In Zaki et al (1996) we show a mechanism which further reduces the memory requirement to only $k \cdot \mathscr{H}$. The approach in the parallel setting yields a total requirement of $k \cdot \mathscr{H} \cdot \mathscr{P}$.

5. Memory Placement Policies for Association Mining

In this section we describe a set of custom memory placement policies for improving the locality and reducing false sharing for parallel association mining. To support the different policy features we implemented a custom memory placement and allocation library. In contrast to the standard Unix malloc library, our library allows greater flexibility in controlling memory placement. At the same time it offers a faster memory freeing option, and efficient reuse of pre-allocated memory. Furthermore it does not pollute the cache with boundary tag information.

5.1. Improving Reference Locality

It was mentioned in the introduction that it is extremely important to reduce the memory latency for data-intensive applications like association mining by effective use of the memory hierarchy. In this section we consider several memory placement policies for the hash tree recursive structure used in the parallel mining algorithm. All the locality-driven placement strategies are directed in the way in which the hash tree and its building blocks (hash tree nodes, hash table, itemset list, list nodes, and the itemsets) are traversed with respect to each other. The specific policies we looked at are described below.

Common candidate partitioned database (CCPD): This is the original program that includes calls to the standard Unix memory allocation library available on the target platform.

Simple placement policy (SPP): This policy does not use the data structure access patterns. All the different hash tree building blocks enumerated above are allocated memory from a single region. This scheme does not rely on any special placement of the blocks based on traversal order. Placement is implicit in the order of hash tree creation. Data structures with consecutive calls to the memory allocation routine are adjacent in memory. The runtime overhead involved for this policy is minimal. There are three possible variations of the simple policy depending on where the data structures reside: (1) *common region*: all data structures are allocated from a single global region; (2) *individual regions*: each data structure is allocated memory from a separate region specific to that structure; and (3) *grouped regions*: data structures are grouped together using program semantics and allocated memory from the same region. The SPP scheme in this paper uses the *common region* approach.

Localized placement policy (LPP): This is an instance of a policy grouping related data structures together using local access information present in a single subroutine. In this policy we utilize a 'reservation' mechanism to ensure that a list node (**LN**) with its associated itemset (**Itemset**) in the leaves of the final hash tree are together whenever possible, and that the hash tree node (**HTN**) and the itemset list header (**ILH**) are together. The rationale behind this placement is the way the hash tree is traversed in the support counting phase. An access to a list node is always followed by an access to its itemset, and an access to a leaf hash tree node is followed by an access to its itemset list header. This contiguous memory placement of the different components is likely to ensure that when one component is brought into the cache, the subsequent component is also brought in, thus improving the cache hit rate and the locality.

Global placement policy (GPP): This policy uses global access information to apply the placement policy. We utilize the knowledge of the entire hash tree traversal order to place the hash tree building blocks in memory, so that the next structure to be accessed lies in the same cache line in most cases. The construction of the hash tree proceeds in a manner similar to SPP. We then remap the entire tree according the tree access pattern in the support counting phase. In our case the hash tree is remapped in a depth-first order, which closely approximates the hash tree traversal. The remapped tree is allocated from a separate region. Remapping costs are comparatively low and this policy also benefits by delete aggregation and the lower maintenance costs of our custom library.

5.1.1. Custom Placement Example

A graphical illustration of the CCPD, simple placement policy, and the global placement policy appears in Fig. 5. The figure shows an example hash tree that has three internal nodes (**HTN#1**, **HTN#10**, **HTN#3**), and two leaf nodes (**HTN#12**, **HTN#5**). The numbers that follow each structure show the creation order when the hash tree is first built. For example, the very first memory allocation is for hash tree node **HTN#1**, followed by the hash table **HTNP#2**, and so on. The very last memory allocation is for **Itemset#14**. Each internal

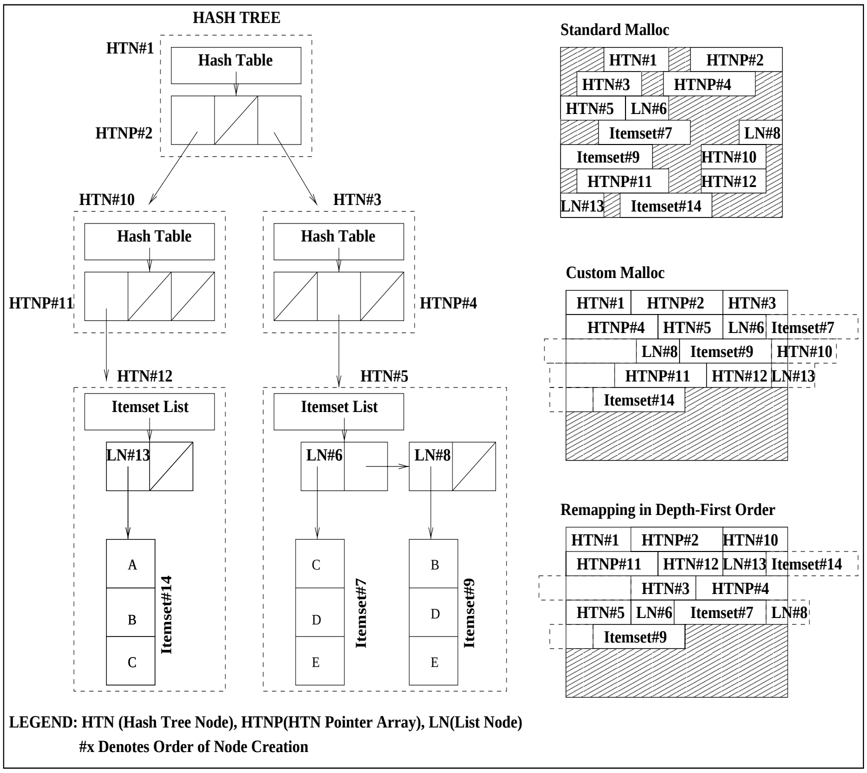

Fig. 5. Improving reference locality via custom memory placement of hash tree.

node points to a hash table, while the leaf nodes have a linked list of candidate itemsets.

CCPD: The original CCPD code uses the standard malloc library. The different components are allocated memory in the order of creation and may come from different locations in the heap. The box on the top right-hand side labeled **Standard Malloc** corresponds to this case.

SPP: The box labeled **Custom Malloc** corresponds to the SPP placement policy. It should be clear from the figure that no access information is used. The allocation order of the components is thus the same as in the CCPD case. However, the custom malloc library ensures that all allocations are contiguous in memory, and can therefore greatly assist cache locality.

GPP: The last box labeled **Remapping in Depth-First Order** corresponds to the GPP policy. In the support counting phase each subset of a transaction is generated in lexicographic order, and a tree traversal is made. This order roughly corresponds to a depth-first search. We use this global access information to remap the different structures so that those most likely to be accessed one after the other are also contiguous in memory. The order of the components in memory is now **HTN#1**, **HTNP#2**, **HTN#10**, **HTNP#11**, etc., corresponding to a depth-first traversal of the hash tree.

5.2. Reducing False Sharing

A problem unique to shared-memory systems is false sharing, which occurs when two different shared variables are located in the same cache block, causing the block to be exchanged between the processors even though the processors are accessing different variables. For example, the support counting phase suffers from false sharing when updating the itemset support counters. A processor has to acquire the lock, update the counter and release the lock. During this phase, any other processor that had cached the particular block on which the lock was placed gets its data invalidated. Since 80% of the time is spent in the support counting step it is extremely important to reduce or eliminate the amount of false sharing. Furthermore, one has to be careful not to destroy locality while improving the false sharing. Several techniques for alleviating this problem are described next.

Padding and aligning: As a simple solution one may place unrelated data that might be accessed simultaneously on separate cache lines. We can thus align the locks to separate cache lines or simply pad out the rest of the coherence block after the lock is allocated. Unfortunately this is not a very good idea for association mining because of the large number of candidate itemsets involved in some of the early iterations (around 0.5 million itemsets). While padding will eliminate false sharing, it will result in unacceptable memory space overhead and, more importantly, a significant loss in locality.

Segregate read-only data: Instead of padding we chose to separate out the locks and counters from the itemset, i.e., to segregate read-only data) (itemsets) from read-write data (locks and counters). All the data in the hash tree that is read-only comes from a separate region and locks and counters come from a separate region. This ensures that read-only data is never falsely shared. Although this scheme does not eliminate false sharing completely, it does eliminate unnecessary invalidations of the read-only data, at the cost of some loss in locality and an additional overhead due to the extra placement cost. To each of the policies for improving locality we added the feature that the locks and the support counters come from a separate region, for reducing false sharing. There are three resulting strategies – L-SPP, L-LPP, and L-GPP – which correspond to the simple placement policy, localized placement policy, and global placement policy, respectively.

Privatize (and reduce): Another technique for eliminating false sharing is called *software caching* (Bianchini and Thomas, 1992) or *privatization*. It involves making a private copy of the data that will be used locally, so that operations on that data do not cause false sharing. For association mining, we can utilize that fact that the support counter increment is a simple addition operation and one that is associative and commutative. This property allows us to keep a local array of counters per processor allocated from a private lock region. Each processor can increment the support of an itemset in its local array during the support counting phase. This is followed by a global sum-reduction. This scheme eliminates false sharing completely, with acceptable levels of memory wastage. This scheme also eliminates the need for any kind of synchronization among processors, but it has to pay the cost of the extra reduction step. We combine this scheme with the global placement policy to obtain a scheme which has good locality and eliminates false

Table 2. Database properties.

Database	T	I	$\mathscr{D}$	Total size
T5.I2.D100K	5	2	100,000	2.6MB
T10.I4.D100K	10	4	100,000	4.3MB
T15.I4.D100K	15	4	100,000	6.2MB
T20.I6.D100K	20	6	100,000	7.9MB
T10.I6.D400K	10	6	400,000	17.1MB
T10.I6.D800K	10	6	800,000	34.6MB
T10.I6.D1600K	10	6	1,600,000	69.8MB
T10.I6.D3200K	10	6	3,200,000	136.9MB

sharing completely. We call this scheme the *local counter array–global placement policy*, (LCA-GPP). The next section presents an experimental evaluation of the different schemes for improving locality and reducing false sharing.

6. Experimental Evaluation

All the experiments were performed on a 12-node SGI Power Challenge SMP machine running IRIX 5.3. Each node is a 100 MHz MIPS processor 16 kB primary cache and 1 MB secondary cache. There is a total of 256 MB of main memory of which around 40 MB is reserved for the OS kernel. The databases are stored on a non-local 2 GB disk and there is exactly one network port for the machine. As a result disk accesses are inherently sequential.

We used different synthetic databases that have been used as benchmark databases for many association rules algorithms (Agrawal et al, 1993; Houtsma and Swami, 1995; Park et al, 1995a; Savasere et al, 1995; Agrawal et al, 1996; Brin et al, 1997; Zaki et al, 1997c; Lin and Kedem, 1998; Lin and Dunham, 1998). The dataset generation procedure is described in Agrawal et al (1996), and the code is publicly available from IBM (http://www.almaden.ibm.com/cs/quest/syndata.html).

These datasets mimic the transactions in a retailing environment, where people tend to buy sets of items together, the so-called potential maximal frequent set. The size of the maximal elements is clustered around a mean, with a few long itemsets. A transaction may contain one or more of such frequent sets. The transaction size is also clustered around a mean, but a few of them may contain many items.

Let D denote the number of transactions, T the average transaction size, I the size of a maximal potentially frequent itemset, L the number of maximal potentially frequent itemsets, and N the number of items. The data is generated using the following procedure. We first generate L maximal itemsets of average size I, by choosing from the N items. We next generate D transactions of average size T by choosing from the L maximal itemsets. We refer the reader to Agrawal and Srikant (1994) for more detail on the database generation. In our experiments we set $N = 1000$ and $L = 2000$. Experiments are conducted on databases with different values of D, T, and I. The database parameters are shown in Table 2. Figure 6 also shows the intermediate hash tree sizes. This indicates to what extent a dataset is amenable to locality placement. In general more improvement is possible for larger trees. Figure 7 shows the number of iterations and the number of frequent itemsets found for different databases. This indicates the complexity

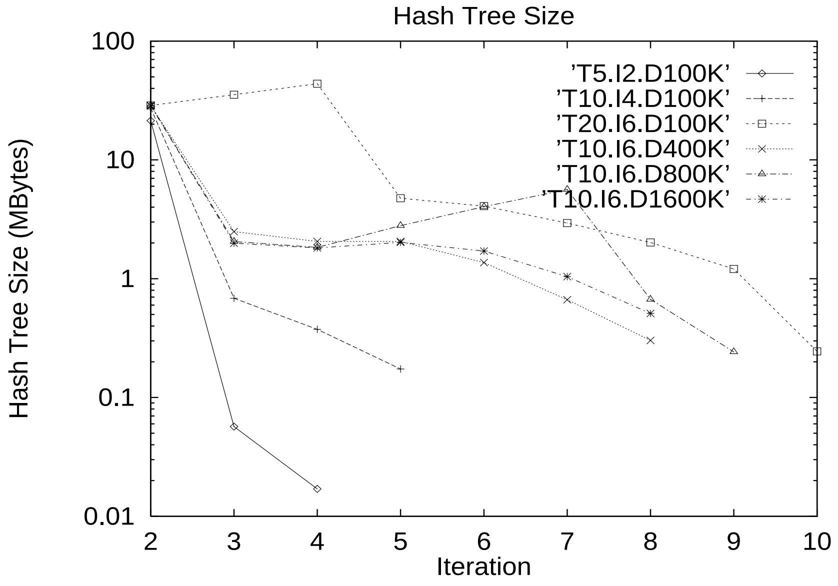

Fig. 6. Intermediate hash tree size (0.1% support).

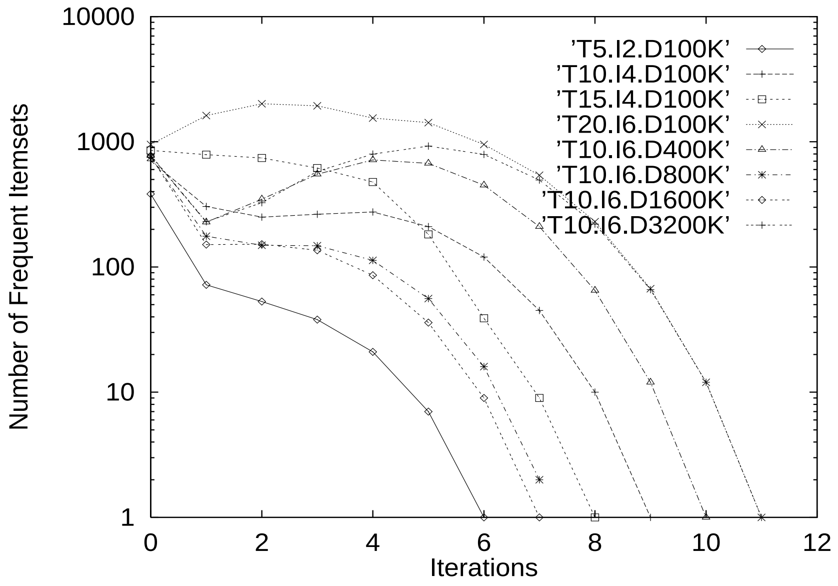

Fig. 7. Frequent itemsets per iteration (0.5% support).

of the dataset. In the following sections all the results are reported for the CCPD parallelization. We do not present any results for the PCCD approach since it performs very poorly, and results in a speed-down on more than one processor. This is because in the PCCD approach every processor has to read the entire

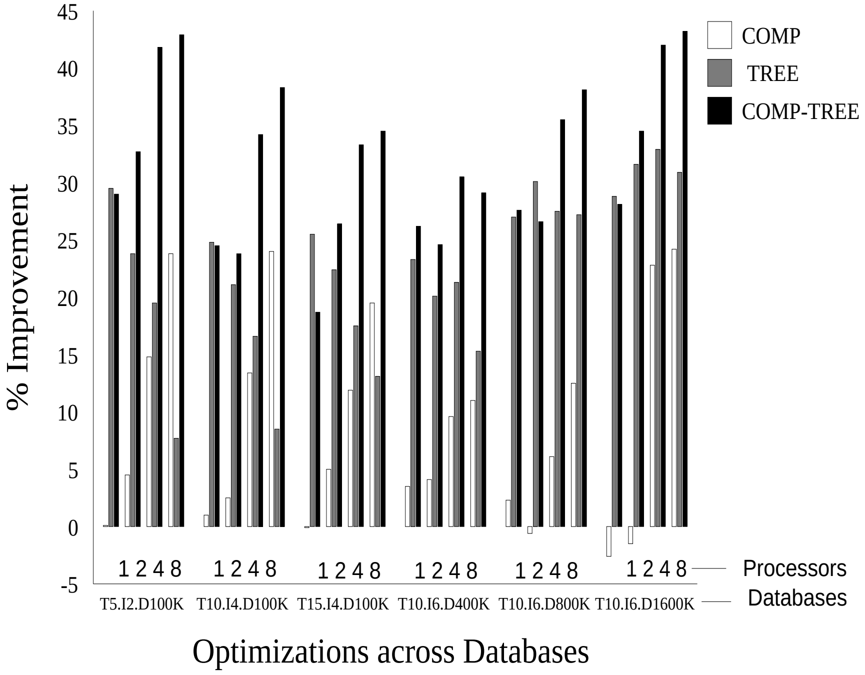

Fig. 8. Effect of computation and hash tree balancing (0.5% support).

database during each iteration. The resulting I/O costs on our system were too prohibitive for this method to be effective.

6.1. Computation and Hash Tree Balancing

Figure 8 shows the improvement in the performance obtained by applying the computation balancing optimization, and the hash tree balancing optimization. The figure shows the percent improvement over a run on an equivalent number of processors without any optimizations. The percent improvements shown in all the experiments are only based on the computation time since we specifically wanted to measure the effects of the optimizations on the computation. Results are presented for different databases and on different number of processors. We first consider only the computation balancing optimization (COMP) using the multiple equivalence classes algorithm. As expected, this doesn't improve the execution time for the uniprocessor case, as there is nothing to balance. However, it is very effective on multiple processors. We get an improvement of around 20% on eight processors. The second column (for all processors) shows the benefit of just balancing the hash tree (TREE) using our bitonic hashing (the unoptimized version uses the simple mod interleaved hash function). Hash tree balancing by itself is an extremely effective optimization. It improves the performance by about 30% even on uni-processors. On smaller databases and eight processors, however, it is not as good as the COMP optimization. The reason is that the hash tree balancing is not sufficient to offset the inherent load imbalance in the candidate generation in this case. The most effective approach is to apply both

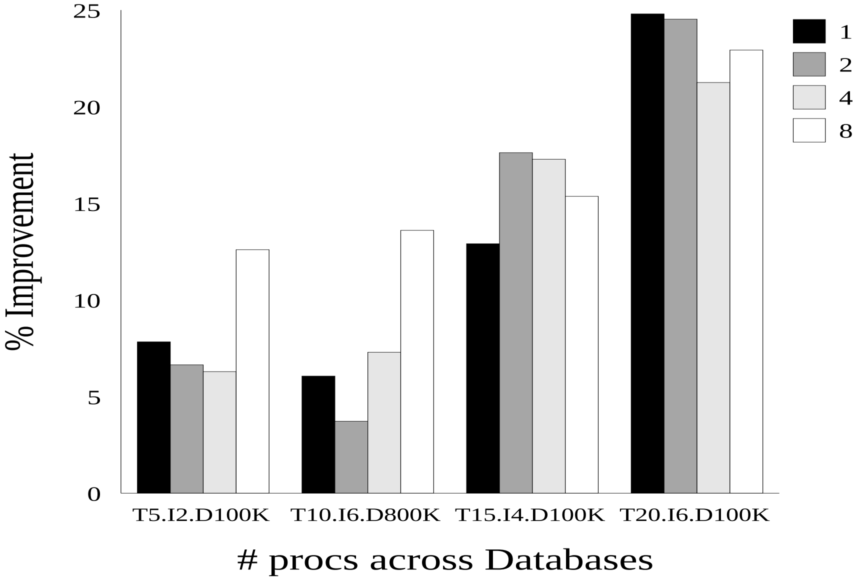

Fig. 9. Effect of short-circuited subset checking (0.5% support).

optimizations at the same time (COMP-TREE). The combined effect is sufficient to push the improvements in the 40% range in the multiple-processor case. On one processor only hash tree balancing is beneficial, since computation balancing only adds extra cost.

6.2. Short-Circuited Subset Checking

Figure 9 shows the improvement due to the short-circuited subset checking optimization with respect to the unoptimized version. The results are presented for different numbers of processors across different databases. The results indicate that while there is some improvement for databases with small transaction sizes, the optimization is most effective when the transaction size is large. In this case we get improvements of around 25% over the unoptimized version.

To gain further insight into this optimization, consider Fig. 10. It shows the percentage improvement obtained per iteration on applying this optimization on the T20.I6.D100K database. It shows results only for the uni-processor case. However, similar results were obtained on more processors. We observe that as the iteration k increases, there is more opportunity for short-circuiting the subset checking, and we get increasing benefits of up to 60%. The improvements start to fall off at the high end where the number of candidates becomes small, resulting in a small hash tree and less opportunity for short-circuiting.

6.3. Parallel Performance

Figure 11 presents the speed-ups obtained on different databases and different processors for the CCPD parallelization. The results presented on CCPD use

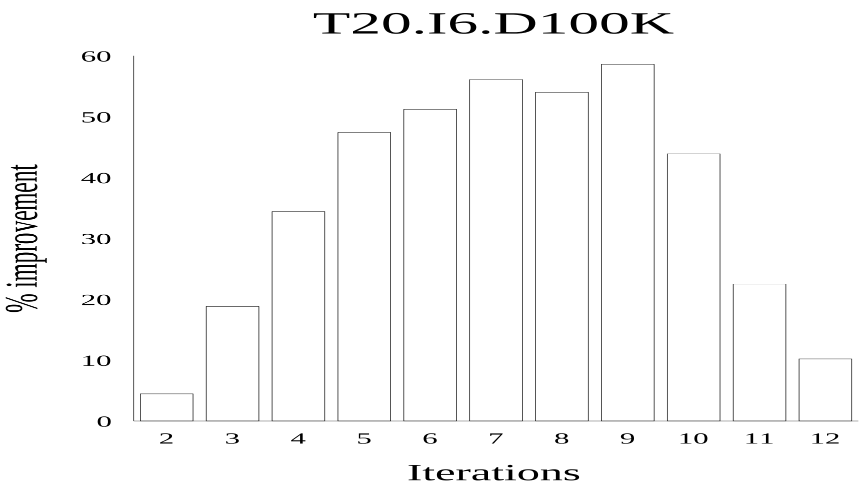

Fig. 10. Percent improvement per iteration (# proc = 1, 0.5% support).

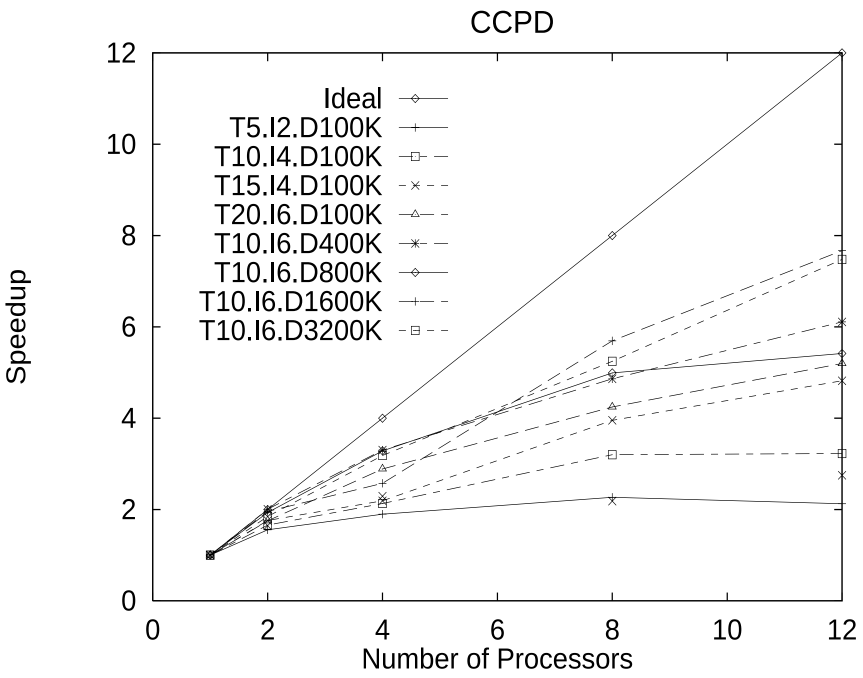

Fig. 11. CCPD: parallel speed-up (0.5% support).

all the optimization discussed in Section 4 – computation balancing, hash tree balancing and short-circuited subset checking. We observe that as the number of transactions increase we get increasing speed-up, with a speed-up of almost eight on 12 processors for the T10.I6.D1600K database, with 1.6 million transactions.

When we look at the break-up of execution time for T5.I2.D100K in the

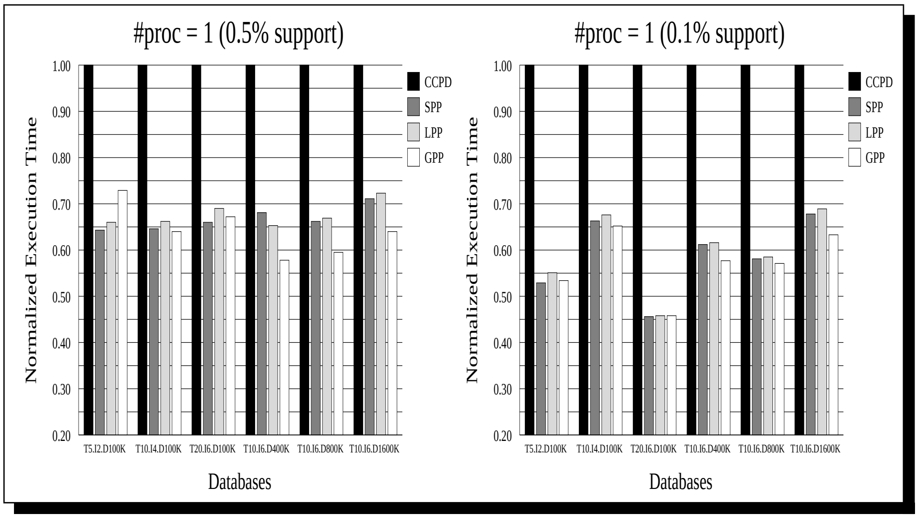

Fig. 12. Memory placement policies: One processor.

sequential case we find that 40% of the time was spent on disk accesses. In our set-up disk accesses are inherently sequential, since all processors are accessing the same non-local disk, so 40% of the execution is inherently non-parallel. This results in an upper bound on speed-up: a shade above two. We obtain a speed-up of around two at four processors.

Similarly on examining the break-up of execution time for T10.I6.D1600K in the sequential case we find that roughly 10% of the time was spent on disk accesses. Again this results in an upper bound on speed-up: a shade under 10. Our algorithm obtains a speed-up of close eight on 12 processors.

We observed similar results for the other datasets as well. Other factors, outside of disk I/O limitations, that play a role in inhibiting speed-up include contention for the bus, false and true sharing for the heap nodes when updating the subset counts and the sequential heap generation phase. Furthermore, since variable length transactions are allowed, and the data is distributed along transaction boundaries, the workload is not uniformly balanced. We next evaluate how we can eliminate some of these overheads while further improving the locality of the algorithm.

6.4. Improving Locality and Reducing False Sharing

6.4.1. Uniprocessor Performance

In the sequential run we compare the three locality enhancing strategies on different databases. It should be clear that there can be no false sharing in a uniprocessor setting. The experimental results are shown in Fig. 12. All times are normalized with respect to the CCPD time for each database. We observe that the simple placement (SPP) strategy does extremely well, with 40–55% improvement over the base algorithm. This is due to the fact that SPP is the least complex in terms of runtime overhead. Furthermore, on a single processor, the creation order of the hash tree is approximately the same as the access order, that is, the candidates are inserted in the tree in lexicographic order. Later in the support counting phase, the subsets of a transaction are also formed in lexicographic order.

The global placement strategy involves the overhead of remapping the entire hash tree (less than 2% of the running time). On smaller datasets we observe that the gains in locality are not sufficient in overcoming this overhead, but as we move to larger datasets the global strategy performs the best. As the size of the dataset increases the locality enhancements become more prominent since more time is spent in the support counting phase of the program. However, we also observe that for decreasing support the size of the intermediate hash tree increases, resulting in an increase in the remapping cost, which undermines locality gains. For example, comparing Fig. 12(a) (0.5% support), and 12(b) (0.1% support), we observe that the gains of GPP over SPP decrease somewhat. Nevertheless, GPP remains the best overall scheme.

6.4.2. Multiprocessor Performance

We now discuss the experimental results for the multiprocessor case, where both the locality and false sharing sensitive placement schemes are important. Figure 13 shows the results for the different placement schemes on four and eight processors. All times are normalized with respect to the CCPD time for each database. In the graphs, the strategies have roughly been arranged in increasing order of complexity from left to right, i.e., CCPD is the simplest, while LCA-GPP is the most complex. The databases are also arranged in increasing order of size from left to right. We observe that with larger databases (both in number of transactions and in the average size of the transaction), the effect of improving locality becomes more apparent. This is due to the fact that larger databases spend more time in the support counting phase, and consequently there are more locality gains. We note that the more complex the strategy, the better it does on larger datasets, since there are enough gains to offset the added overhead.

On comparing the bar charts for the 0.5% support and 0.1% we note that the relative benefits of our optimizations become more apparent on lower support (for the T20.I6.D100K database the relative benefits for 0.1% support case is 30% more than for the 0.5% support case). It is also interesting to note that for 0.5% support the only database in which GPP performs better than SPP is the largest database in our experiments, whereas for 0.1% support GPP performs better than SPP for all except the two smallest databases. Both these observations can be attributed to the fact that the ratio of memory placement overhead to the total execution time is higher for higher support values. Further, on going from a lower support (0.1%) to a higher one (0.5%), the net effective benefit of placement also reduces with reduction in hash tree size.

We will now discuss some policy-specific trends. The base CCPD algorithm suffers from a poor reference locality due to the dynamic nature of memory allocations. It also suffers from high amounts of false sharing, due to the shared accesses to the itemsets, locks and support counters. The simple placement policy, SPP, which places all the hash tree components contiguously, performs extremely well. This is not surprising since it imposes the smallest overhead of all the special placement strategies, and the dynamic creation order to an extent matches the hash tree traversal order in the support counting phase. We thus obtain a gain of 40–60% by using the simple scheme alone. When SPP is augmented with separate locks and counters region, i.e., for L-SPP, we obtain slightly better results than SPP. As with all lock-region based schemes, L-SPP loses on locality, but with

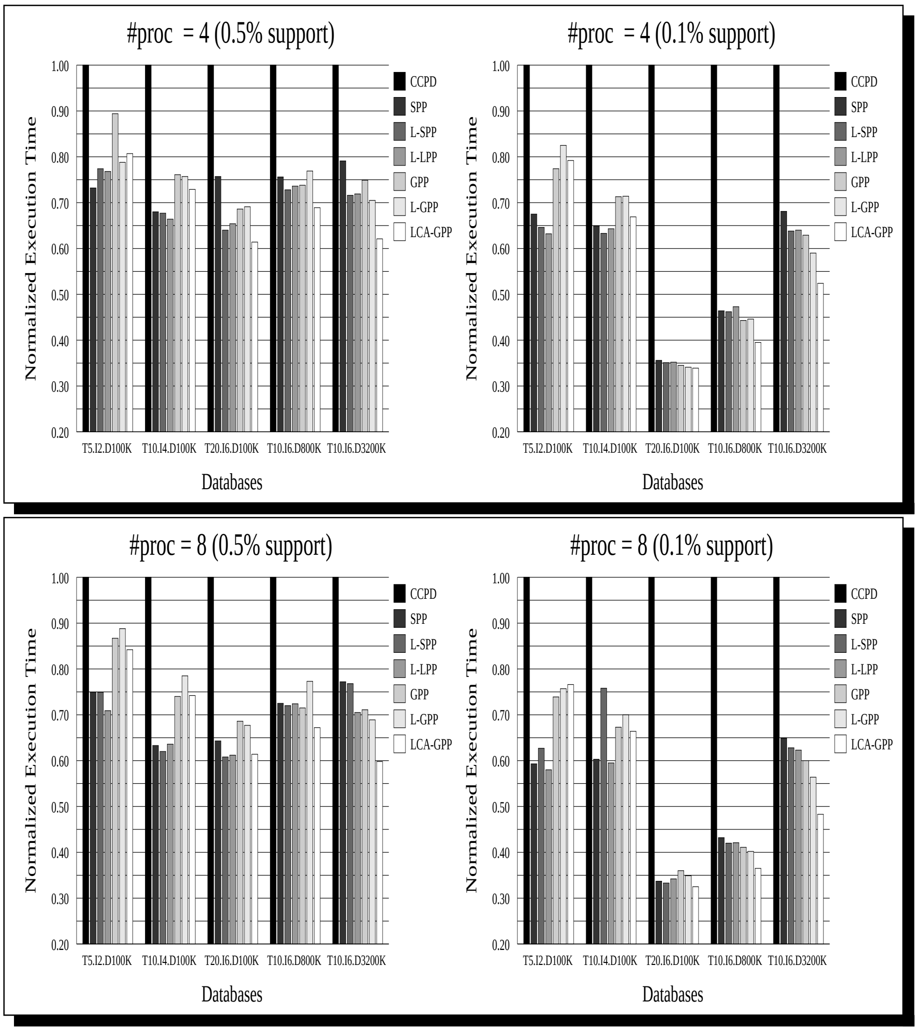

Fig. 13. Memory placement policies: Four and eight processors.

increasing datasets the benefits of reducing false sharing outweigh the overhead. The localized placement of L-LPP doesn't help much. It is generally very close to L-SPP. From a pure locality viewpoint, the global placement policy performs the best with increasing data size. It uses the hash tree traversal order to rearrange the hash tree to maximize locality, resulting in significant performance gains. Reducing the false sharing of the GPP policy by coupling it with a separate lock-region in L-GPP does worse than GPP on the smaller datasets due to its increased overhead. However, for the biggest database, it outperformed the previous policies. The best overall scheme is LCA-GPP, which has a local counter array per processor. It eliminates false sharing and synchronization completely, but it retains good locality. It thus performs the best for the large datasets.

7. Related Work

7.1. Association Mining

7.1.1. Sequential Algorithms

Several algorithms for mining associations have been proposed in the literature (Agrawal et al, 1993; Manila et al, 1994; Houtsma and Swami, 1995; Mueller, 1995; Park et al, 1995a; Savasere et al, 1995; Agrawal et al, 1996; Toivonen, 1996; Brin et al, 1997; Lin and Kedem, 1998; Lin and Dunham, 1998). The *Apriori* algorithm (Agrawal et al, 1996) is the best-known previous algorithm, and it uses an efficient candidate generation procedure, such that only the frequent itemsets at a level are used to construct candidates at the next level. However, it requires multiple database scans. The DHP algorithm (Park et al, 1995a) tries to reduce the number of candidates by collecting approximate counts in the previous level. Like *Apriori* it requires as many database passes as the longest itemset. The *Partition* algorithm (Savasere et al, 1995) minimizes I/O by scanning the database only twice. It partitions the database into small chunks which can be handled in memory. In the first pass it generates the set of all potentially frequent itemsets, and in the second pass it counts their global support. The DLG (Yen and Chen, 1996) algorithm uses a bit-vector per item, noting the transactions where the item occurred. It generates frequent itemsets via logical AND operations on the bit-vectors. However, DLG assumes that the bit vectors fit in memory, and thus scalability could be a problem for databases with millions of transactions. The DIC algorithm (Brin et al, 1997) dynamically counts candidates of varying length as the database scan progresses, and thus is able to reduce the number of scans. Another way to minimize the I/O overhead is to work with only a small sample of the database. An analysis of the effectiveness of sampling for association mining was presented in Zaki et al (1997b), and Toivonen (1996) presents an exact algorithm that finds all rules using sampling. The AS-CPA algorithm and its sampling versions (Lin and Dunham, 1998) build on top of *Partition* and produce a much smaller set of potentially frequent candidates. It requires at most two database scans. Also, sampling may be used to eliminate the second pass altogether. Approaches using only general-purpose DBMS systems and relational algebra operations have also been studied (Holsheimer et al, 1995; Houtsma and Swami, 1995).

All the above algorithms generate all possible frequent itemsets. Methods for finding the maximal elements include *All-MFS* (Gunopulos et al, 1997), which is a randomized algorithm to discover maximal frequent itemsets. The *Pincer-Search* algorithm (Lin and Kedem, 1998) not only constructs the candidates in a bottom-up manner like *Apriori*, but also starts a top-down search at the same time. This can help in reducing the number of database scans. *MaxMiner* (Bayardo, 1998) is another algorithm for finding the maximal elements. It uses efficient pruning techniques to quickly narrow the search space. We recently proposed new association mining algorithms that usually make only three scans (Zaki et al, 1997c, 1997d). They use novel itemset clustering techniques, based on equivalence classes and maximal hypergraph cliques, to approximate the set of potentially maximal frequent itemsets. Efficient lattice traversal techniques based on bottom-up and hybrid search, are then used to generate the frequent itemsets contained in each cluster. These algorithms (Zaki et al, 1997c, 1997d) range from those that generate all frequent itemsets to those that generate the maximal frequent itemsets.

7.1.2. Parallel Algorithms

Distributed-memory machines: Three different parallelizations of *Apriori* on IBM-SP2, a distributed-memory machine, were presented in Agrawal and Shafer (1996). The *Count Distribution* algorithm is a straightforward parallelization of *Apriori*. Each processor generates the partial support of all candidate itemsets from its local database partition. At the end of each iteration the global supports are generated by exchanging the partial supports among all the processors. The *Data Distribution* algorithm partitions the candidates into disjoint sets, which are assigned to different processors. However, to generate the global support each processor must scan the entire database (its local partition, and all the remote partitions) in all iterations. It thus suffers from huge communication overhead. The *Candidate Distribution* algorithm also partitions the candidates, but it selectively replicates the database, so that each processor proceeds independently. The local database portion is still scanned in every iteration. *Count Distribution* was shown to have superior performance among these three algorithms (Agrawal and Shafer, 1996). Other parallel algorithms improving upon these ideas in terms of communication efficiency or aggregate memory utilization have also been proposed (Cheung et al, 1996a, 1996b; Han et al, 1997). The PDM algorithm (Park et al, 1995b) presents a parallelization of the DHP algorithm (Park et al, 1995a). The hash-based parallel algorithms NPA, SPA, HPA, and HPA-ELD, proposed in Shintani and Kitsuregawa (1996) are similar to those in Agrawal and Shafer (1996). Essentially NPA corresponds to Count Distribution, SPA to Data Distribution, and HPA to Candidate Distribution. The HPA-ELD algorithm is the best among NPA, SPA, and HPA, since it eliminates the effect of data skew, and reduces communication by replicating candidates with high support on all processors.

In recent work we presented new parallel association mining algorithms (Zaki et al, 1997e). They utilize the fast sequential algorithms proposed in Zaki et al (1997a, 1997c) as the base algorithm. The database is also selectively replicated among the processors so that the portion of the database needed for the computation of associations is local to each processor. After the initial set-up phase, the algorithms do not need any further communication or synchronization. The algorithms minimize I/O overheads by scanning the local database portion only twice. Furthermore they have excellent locality since only simple intersection operations are used to compute the frequent itemsets.

Shared-memory machines: To the best of our knowledge the CCPD was the first algorithm targeting shared-memory machines. In this paper we presented the parallel performance of CCPD, and also the benefits of optimizations like hash tree balancing, parallel candidate generation, and short-circuited subset checking. We further studied the locality and false sharing problems encountered in CCPD, and solutions for alleviating them. Parts of this paper have appeared in Zaki et al (1996) and Parthasarathy et al (1998). A recent paper presents APM (Chung et al, 1998), an asynchronous parallel algorithm for shared-memory machines based on the DIC algorithm (Brin et al, 1997). Our proposed optimizations, and locality enhancing and false sharing reducing policies, are orthogonal to their approach.

7.2. Improving Locality

Several automatic techniques like tiling, strip-mining, loop interchange and uniform transformations (Anderson and Lam, 1993; Carr et al, 1994; Cierniak and

Li, 1995; Li, 1995) have been proposed on a wide range of architectures, to improve the locality of array-based programs. However, these cannot directly be applied for dynamic data structures. *Prefetching* is also a suggested mechanism to help tolerate the latency problem. Automatic prefetching based on locality on array-based programs was suggested in work done in Mowry et al (1992). Building upon this work, more recently in Luk and Mowry (1996), a compiler-based prefetching strategy on recursive data structures was presented. They propose a data linearization scheme, which linearizes one data class in memory to support prefetching, whereas we suggest an approach where related structures are grouped together based on access patterns to further enhance locality. Our experimental results indicate increased locality gains when grouping related data structures, rather than linearizing a single class.

7.3. Reducing False Sharing

Five techniques mainly directed at reducing false sharing were proposed in Torrellas et al (1990). They include *padding*, *aligning*, and allocation of memory requested by different processors from different heap regions. Those optimizations result in a good performance improvement for the applications they considered. In our case study padding and aligning were not found to be very beneficial. Other techniques to reduce false sharing on array-based programs include *indirection* (Eggers and Jeremiassen, 1991), *software caching* (Bianchini and Thomas, 1992), and *data remapping* (Anderson et al, 1995). Our placement policies have utilized some of these techniques.

7.4. Memory Allocation

General-purpose algorithms for dynamic storage have been proposed and honed for several years (Knuth, 1973; Kingsley, 1982; Weinstock and Wulf, 1988). An evaluation of the performance of contemporary memory allocators on five allocation-intensive C programs was presented in Grunwald et al (1993). Their measurements indicated that a custom allocator can be a very important optimization for memory subsystem performance. They pointed out that most of the extant allocators suffered from cache pollution due to maintenance of boundary tags and that optimizing for space (minimizing allocations) can serve the opposite end of reducing speed, due to lower locality. All of these points were borne out by our experimental results and went into the design of our custom memory placement library. Our work differs from theirs in that they rely on custom allocation, while we rely on custom memory placement (where related data are placed together) to enhance locality and reduce false sharing.

8. Conclusions

In this paper, we presented a parallel implementation of the *Apriori* algorithm on the SGI Power Challenge shared memory multi-processor. We discussed a set of optimizations which include optimized join and pruning, computation balancing for candidate generation, hash tree balancing, and short-circuited subset checking. We then presented experimental results on each of these. Improvements of more than 40% were obtained for the computation and hash tree balancing. The short-circuiting optimization was found to be effective for databases with large transaction sizes. We also achieved good speed-ups for the parallelization.

Locality and false sharing are important issues in modern multiprocessor machines due to the increasing gap between processor and memory subsystem performance – particularly so in data-mining applications, such as association mining, which operate on large sets of data and use large pointer-based dynamic data structures, where memory performance is critical. Such applications typically suffer from poor data locality and high levels of false sharing where shared data is written. In this paper, we proposed a set of policies for controlling the allocation and memory placement of such data structures, to achieve the conflicting goal of maximizing locality, while minimizing the level of false sharing for such applications.

Our experiments show that simple placement schemes can be quite effective, and for the datasets we looked at they improve the execution time of our application by up to a factor of two (50% improvement over the base case). We also observed that as the datasets became larger (alternately at lower support values), and there is more work per processor the more complex placement schemes improve matters further (up to 20% for the largest example). It is likely that for larger datasets this gain from more complex placement schemes will increase. While we evaluated these policies for association mining, the proposed techniques are directly applicable to other mining algorithms such as quantitative associations, multi-level associations, and sequential patterns.

Acknowledgements. This work was supported in part by an NSF Research Initiation Award (CCR-9409120) and ARPA contract F19628-94-C-0057.

References

Agrawal R, Manila H, Srikant R, et al (1996) Fast discovery of association rules. In Fayyad U et al (eds). Advances in knowledge discovery and data mining. AAAI Press, Menlo Park, CA, pp 307–328

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

Agrawal R, Srikant R (1994) Fast algorithms for mining association rules. In 20th VLDB conference, September 1994

Agrawal R, Srikant R (1995) Mining sequential patterns. In 11th international conference on Data Engineering

Agrawal R, Imielinski T, Swami A (1993) Mining association rules between sets of items in large databases. In ACM SIGMOD conference on management of data, May 1993

Anderson JM, Lam M (1993) Global optimizations for parallelism and locality on scalable parallel machines. In ACM conference n programming language design and implementation, June 1993

Anderson JM, Amarsinghe SP, Lam M (1995) Data and computation transformations for multiprocessors. In ACM symposium on principles and practice of parallel programming

Bayardo RJ (1998) Efficiently mining long patterns from databases. In ACM SIGMOD conference on management of data, June 1998

Bianchini R, LeBlanc TJ (1992) Software caching on cache-coherent multiprocessors. In 4th symposium on parallel distributed processing, December 1992, pp 521–526

Brin S, Motwani R, Ullman J, Tsur S (1997) Dynamic itemset counting and implication rules for market basket data. In ACM SIGMOD conference on management of data, May 1997

Carr S, McKinley KS, Tseng C-W (1994) Compiler optimizations for improving data locality. In 6th international conference on architectural support for programming languages and operating systems, October 1994, pp 252–262

Cheung D, Han J, Ng V et al (1996) A fast distributed algorithm for mining association rules. In 4th International conference on parallel and distributed information systems, December 1996

Cheung D, Hu K, Xia S (1998) Asynchronous parallel algorithm for mining association rules on shared-memory multi-processors. In 10th ACM symposium parallel algorithms and architectures, June 1998

Cheung D, Ng V, Fu A, Fu Y (1996) Efficient mining of association rules in distributed databases. IEEE Transactions on Knowledge and Data Engineering 8(6):911–922

Cierniak M, Li W (1995) Unifying data and control transformations for distributed shared-memory machines. In ACM conference on programming language design and implementation, June 1995

Cierniak M, Zaki MJ, Li W (1997) Compile-time scheduling algorithms for a heterogeneous network of workstations. Computer Journal 40(6):356–372

Eggers SJ, Jeremiassen TE (1991) Eliminating false sharing. In International conference on parallel processing, August 1991, pp 377–381

Grunwald D, Zorn B, Henderson R (1993) Improving the cache locality of memory allocation. In ACM conference on programming language design and implementation, June 1993

Gunopulos D, Manila H, Saluja S (1997) Discovering all the most specific sentences by randomized algorithms. In International conference on database theory, January 1997

Han E-H, Karypis G, Kumar V (1997) Scalable parallel data mining for association rules. In ACM SIGMOD conference on management of data, May 1997

Hennessey J, Patterson D (1995) Computer architecture: a quantitative approach. Morgan-Kaufmann, San Mateo, CA

Holsheimer M, Kersten M, Manila H, Toivonen H A perspective on databases and data mining. In 1st International conference on knowledge discovery and data mining, August 1995

Houtsma M, Swami A Set-oriented mining of association rules in relational databases. In 11th International conference on data engineering

Kingsley C (1982) Description of a very fast storage allocator. Documentation of 4.2 BSD Unix malloc implementation, February 1982

Knuth DE (1973) Fundamental algorithms: the art of computer programming (vol 1). Addison-Wesley, Reading, MA

Li W (1995) Compiler cache optimizations for banded matrix problems. In International conference on supercomputing, July 1995, pp 21–30

Lin D-I, Kedem ZM (1998) Pincer-search: a new algorithm for discovering the maximum frequent set. In 6th International conference on extending database technology, March 1998

Lin J-L, Dunham MH (1998) Mining association rules: anti-skew algorithms. In 14th International conference on data engineering, February 1998

Luk C-K, Mowry TC (1996) Compiler-based prefetching for recursive data structures. In 6th International conference on architectural support for programming languages and operating systems

Manila H, Toivonen H, Verkamo I (1994) Efficient algorithms for discovering association rules. In ASIA workshop on knowledge discovery in databases, July 1994

Mowry TC, Lam MS, Gupta A (1992) Design and evaluation of a compiler algorithm for prefetching. In 5th International conference on architectural support for programming languages and operating systems, October 1992, pp 62–73

Mueller A (1995) Fast sequential and parallel algorithms for association rule mining: a comparison. Technical report CS-TR-3515, University of Maryland, College Park, August 1995

Park JS, Chen M, Yu PS (1995a) An effective hash based algorithm for mining association rules. In ACM SIGMOD International conference on management of data, May 1995

Park JS, Chen M, Yu PS (1995b) Efficient parallel data mining for association rules. In ACM International conference on information and knowledge management, November 1995

Parthasarathy S, Zaki MJ, Li W (1998) Memory placement techniques for parallel association mining. In 4th International conference on knowledge discovery and data mining, August 1998

Savasere A, Omiecinski E, Navathe S (1995) An efficient algorithm for mining association rules in large databases. In 21st VLDB conference

Shintani T, Kitsuregawa M (1996) Hash based parallel algorithms for mining association rules. In 4th International conference on parallel and distributed information systems, December 1996

Srikant R, Agrawal R (1995) Mining generalized association rules. In 21st VLDB conference

Srikant R, Agrawal R (1996) Mining quantitative association rules in large relational tables. In ACM SIGMOD conference on management of data, June 1996

Toivonen H (1996) Sampling large databases for association rules. In 22nd VLDB conference

Torrellas J, Lam MS, Hennessy JL (1990) Shared data placement optimizations to reduce multiprocessor cache miss rates. In International conference on parallel processing, August 1990, vol II, pp 266–270

Weinstock CB, Wulf WA (1988) An efficient algorithm for heap storage allocation. In ACM SIGPLAN Notices 23(10):141–148

Yen S-J, Chen ALP (1996) An efficient approach to discovering knowledge from large databases. In 4th International conference on parallel and distributed information systems, December 1996

Zaki MJ, Ogihara M, Parthasarathy S, Li W (1996) Parallel data mining for association rules on shared-memory multi-processors. In Supercomputing'96, November 1996

Zaki MJ, Parthasarathy S, Li W (1997a) A localized algorithm for parallel association mining. In 9th ACM symposium on parallel algorithms and architectures, June 1997

Zaki MJ, Parthasarathy S, Li W, Ogihara M (1997b) Evaluation of sampling for data mining of association rules. In 7th International workshop on research issues in data engineering, April 1997

Zaki MJ, Parthasarathy S, Ogihara M, Li W (1997c) New algorithms for fast discovery of association rules. In 3rd International conference on knowledge discovery and data mining, August 1997

Zaki MJ, Parthasarathy S, Ogihara M, Li W (1997d) New algorithms for fast discovery of association rules. Technical report URCS TR 651, University of Rochester, April 1997

Zaki MJ, Parthasarathy S, Ogihara M, Li W (1997e) Parallel algorithms for fast discovery of association rules. In Data Mining and Knowledge Discovery (special issue on scalable high-performance computing for KDD) 1(4):343–373 .

Author Biographies

Srinivasan Parthasarathy is currently an assistant professor of computer science at the Ohio State University. He received an M.S. degree in electrical engineering from the University of Cincinnati in 1994, and an M.S. and Ph.D. in computer science from the University of Rochester in 1996 and 1999, respectively. His research interests include parallel and distributed systems and data mining.

Mohammed J. Zaki received his Ph.D. in computer science from the University of Rochester in 1998. He is currently an assistant professor of computer science at Rensselaer Polytechnic Institute. His research interests focus on developing efficient parallel algorithms for various data mining and knowledge discovery tasks.

Mitsunori Ogihara (also known as Ogiwara) received a Ph.D. in information sciences from Tokyo Institute of Technology in 1993. He is currently an associate professor of computer science at the University of Rochester. His research interests are computational complexity, DNA computing, and data mining.

Wei Li received his Ph.D. in computer science from Cornell University in 1993. He is currently with Intel Corporation. Before that, he was an assistant professor at the University of Rochester. His current technical interests include Java compilation, software for network computing, and data mining.

Correspondence and offprint requests to: M. J. Zaki, Computer Science Department, Rensselaer Polytechnic Institute, Troy, NY 12180-3590, USA. Email: zaki@cs.rpi.edu