Memory Placement Techniques for Parallel Association Mining *

Srinivasan Parthasarathy, Mohammed J. Zaki, and Wei Li
Computer Science Department, University of Rochester, Rochester NY 14627
{srini,zaki,wei}@cs.rochester.edu

Abstract

Many data mining tasks (e.g., Association Rules, Sequential Patterns) use complex pointer-based data structures (e.g., hash trees) that typically suffer from sub-optimal *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.

Introduction

Improving the *data locality* (by keeping data local to a processor's cache) and reducing/eliminating *false sharing* (by reducing cache coherence operations on non-shared objects) are important issues in modern shared-memory multiprocessors (SMPs) due to the increasing gap between processor and memory subsystem performance (Hennessey 95). Particularly more so in data mining applications, such as association mining (Agrawal 96), which operate on large sets of data and use large pointer based dynamic data structures (such as hash trees, lists, etc.), where memory performance is critical. Such applications typically suffer from poor data locality and high levels of false sharing where shared data is written. Traditional locality optimizations found in the literature (Carr 94; Anderson 93) are only applicable to array-based programs, although some recent work has looked at dynamic structures (Luk 96). Techniques for reducing false sharing have also been studied (Torrellas 90; Bianchini 92; Anderson 95). However, the arbitrary memory allocation in data mining applications makes detecting and eliminating false sharing a very difficult proposition.

In this paper we describe techniques for improving locality and reducing false sharing for parallel association mining. We propose a set of policies for controlling the allocation and memory placement of dynamic data structures so that data likely to be accessed together is grouped together (memory placement) while minimizing false sharing. Our experiments 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 96), multi-level (taxonomies) associations (Srikant 95), and sequential patterns (Agrawal 95), which also use hash tree based structures. Most of the current work in parallel association mining has only focused on distributed-memory machines (Park 1995; Agrawal 96; Cheung 96; Shintani 96; Han 97; Zaki 97), where false sharing doesn't arise. However, our locality optimizations are equally applicable to these methods. A detailed version of this paper appears in (Parthasarathy 97).

*Supported in part by NSF grants CCR-9409120 and CCR-9705594, and ARPA contract F19628-94-C-0057.

Association Mining: Problem Formulation

Let $\mathcal{I}$ be a set of items, and $\mathcal{T}$ a database of transactions, where each transaction has a unique identifier and contains a set of items, called an *itemset*. The *support* of an itemset X, denoted $\sigma(X)$, is the number of transactions in which it occurs as a subset. An itemset is *frequent* if its support is more than a user-specified *minimum support (min_sup)* value. An *association rule* is an expression $A \Rightarrow B$, where A and B are itemsets. The support of the rule is given as $\sigma(A \cup B)$, and the *confidence* as $\sigma(A \cup B)/\sigma(A)$ (i.e., conditional probability of B given that A occurs in a transaction).

The association mining task consists of two steps: 1) Find all frequent itemsets. This step is computationally and I/O intensive, and the main focus of this paper. 2) Generate high confidence rules. This step is relatively straightforward; rules of the form $X \backslash Y \rightarrow Y$ (where $Y \subset X$), are generated for all frequent itemsets X, provided the rules have at least *minimum confidence (min_conf)*.

Parallel Association Mining on SMP Systems

The *Common Candidate Partitioned Database* (CCPD) (Zaki 96) shared-memory algorithm is built on top of the *Apriori* algorithm (Agrawal 96). During iteration 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 generating a candidate set for the next iteration. A pruning step eliminates any candidate which has an infrequent subset. This iterative process is repeated for $k = 1, 2, 3 \cdots$, until there are no more frequent k-itemsets to be found.

The candidates are stored in a hash tree to facilitate fast support counting. An internal node of the hash tree at depth

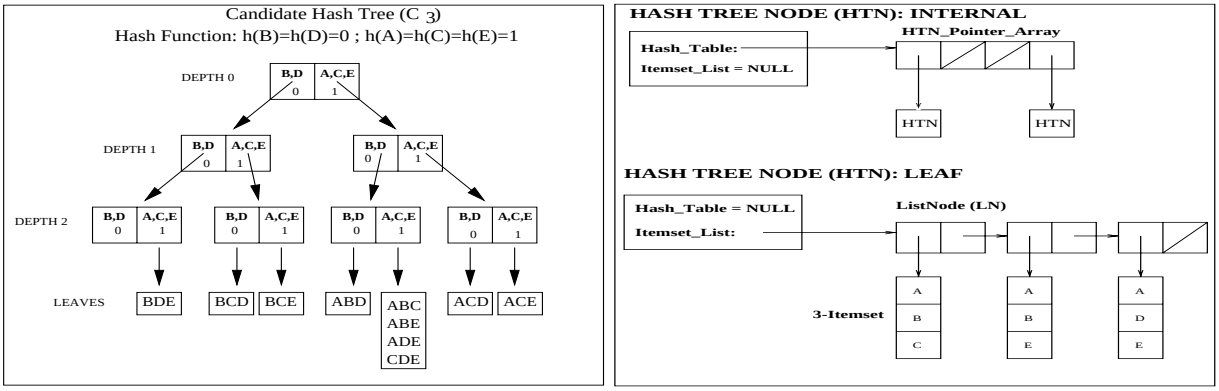

Figure 1: a) Candidate Hash Tree; b) Structure of Internal and Leaf Nodes

d contains a hash table whose cells point to nodes at depth $d+1$. All the itemsets are stored in the leaves in a sorted linked-list. The maximum depth of the tree in iteration k is k. Figure 1a shows a hash tree containing candidate 3-itemsets, and Figure 1b 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 (ISET). 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. To count the support of candidate k-itemsets, for each transaction T in the database, we form all k-subsets of T in lexicographical order. For each subset we then traverse the hash tree, look for a matching candidate, and update its count.

CCPD keeps a common candidate hash tree, but logically splits the database among the processors. The new candidates are generated and inserted into the hash tree in parallel. After candidate generation each processor scans its local database portion and counts the itemset support in parallel. Each itemset has a single counter protected by a lock to guarantee mutual exclusion when multiple processors try to increment it at the same time. Finally, the master process selects the frequent itemsets. The support counting step typically accounts for 80% of the total computation time (Zaki 96). It can be observed that once the tree is built it is accessed multiple times during each iteration (once per transaction). Our locality enhancement techniques focus on improving the performance of these accesses.

Memory Placement Policies

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 preallocated memory. Furthermore it does not pollute the cache with boundary tag information.

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 candidate hash tree. 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.

The *Common Candidate Partitioned Database* (CCPD) scheme is the original program that includes calls to the standard Unix memory allocation library on the target platform.

In the *Simple Placement Policy* (SPP) all the different hash tree building blocks 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 calling 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.

The *Localized Placement Policy* (LPP) groups 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 (ISET) 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. Thus contiguous memory placement of the different components is likely to ensure that when one component is brought into the cache, the subsequent one is also brought in, improving the cache hit rate and locality.

The *Global Placement Policy* (GPP) utilizes 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 approx-

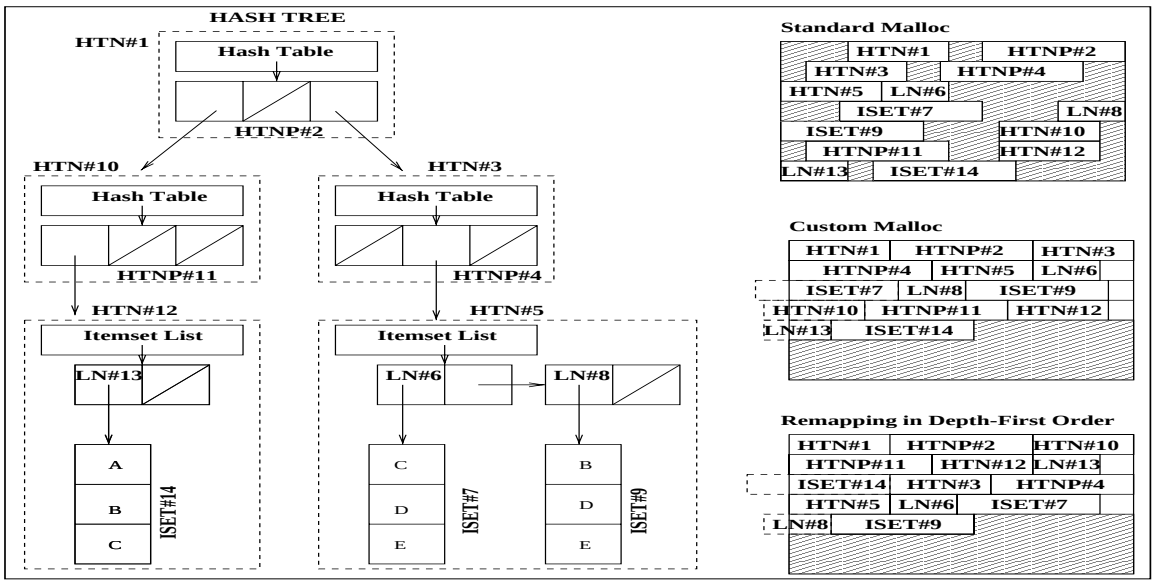

Figure 2: Improving Reference Locality via Custom Memory Placement of Hash Tree

imates 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.

Custom Placement Example A graphical illustration of the CCPD, Simple Placement Policy, and the Global Placement Policy appears in Figure 2. 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 after each structure show the creation order when the hash tree is 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 ISET#14. Each internal node points to a hash table, while the leaf nodes have a linked list of candidate itemsets.

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 top right labeled *Standard Malloc* corresponds to this case. The box labeled *Custom Malloc* corresponds to the SPP policy. No access information is used, and the allocation order of the components is 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. 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.

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 time, 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. Also, one has to be careful not to destroy locality while improving the false sharing. Several techniques for alleviating this problem are described next.

One solution is *Padding and Aligning*, by placing unrelated read-write data on separate cache lines. For example, we can align each itemset lock to a separate cache lines and pad out the rest of the line. 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 eliminates false sharing, it results in unacceptable memory space overhead and more importantly, a significant loss in locality.

In *Segregate Read-Only Data*, instead of padding we separate out the locks and counters (read-write data) from the itemset (read-only data). All the read-only data 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. On top of each of the locality enhancing schemes, we added a separate read-write region for locks and counters. The three resulting strategies are, L-SPP, L-LPP, and L-GPP.

The *Privatize (and Reduce)* scheme 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 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. 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 lock 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 sharing completely. We call this scheme *Local Counter Array-Global Placement Policy* (LCA-GPP).

Experimental Evaluation

All the experiments were performed on a 12-node SGI Challenge SMP machine with 256MB main-memory, running IRIX 5.3. Each node is a 100MHz MIPS processor with a 16KB primary cache and 1MB secondary cache. The databases are stored on a non-local 2GB disk. We used different benchmark databases generated using the procedure described in (Agrawal 96). Table 1 shows the databases used and the total execution times of CCPD on 0.1% (0.5%) minimum support. For example, T5.I2.D100K corresponds to a database with average transaction size of 5, average maximal potential large itemset size of 2, and 100,000 transactions. We set the number of maximal potentially large itemsets $|L| = 2000$, and the number of items $N = 1000$.

Database ($\times$ 100K)	Size (MB)	CCPD 0.1%(0.5%) in Seconds		
		P=1	P=4	P=8
T5.I2.D1	3	44(13)	28(9)	27(8)
T10.I4.D1	4	133(66)	70(32)	57(24)
T20.I6.D1	8	3027(325)	1781(126)	1397(81)
T10.I6.D4	17	895(226)	398(81)	289(56)
T10.I6.D8	35	2611(463)	1383(159)	999(95)
T10.I6.D16	70	2831(884)	1233(280)	793(161)
T10.I6.D32	137	6085(2144)	2591(815)	1677(461)

Table 1: Total Execution Times for CCPD

Figure 3 shows the sequential (P=1) and parallel (P=4,8) performance of the different placement policies on the different databases. 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 have also been arranged in increasing order of size from left to right.

Uniprocessor Performance In the sequential run we only compare the three locality enhancing strategies on different databases, since there can be no false sharing in a uniprocessor setting. The simple placement (SPP) strategy does extremely well, with 35-55% improvement over the base algorithm. This is due to the fact that SPP is the least complex in terms of runtime overhead. Furthermore, one a single processor, the creation order of the hash tree in the candidate generation phase, is approximately the same as the access order in the support counting phase, i.e., in lexicographic order. The global placement strategy (GPP) 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. However GPP performs the best on larger datasets.

Multiprocessor Performance For the multiprocessor case both the locality and false sharing sensitive placement schemes are important. We observe that as the databases become larger, the more complex strategies perform the best. This is due to the fact that larger databases spend more time in the support counting phase, and consequently there are more locality gains, which are enough to offset the added overhead. We also observed that on lower values of support (0.1% vs 0.5% in Figure 3), our optimizations perform better since the ratio of memory placement overhead to the total execution time reduces. Further, on lower support the hash tree tends to be larger, and has greater potential to benefit from appropriate placement. For example, on T20.I6.D100K the relative benefits on 0.1% support are 30% more than on 0.5% support. Also on 0.5% GPP performs better than SPP only on the largest dataset, whereas on 0.1% GPP performs better for all except the 2 smallest datasets.

We will now discuss some policy specific trends. The base CCPD algorithm suffers from a poor reference locality and high amount of false sharing, due to the dynamic nature of memory allocations and due to the shared accesses to the itemsets, locks and counters, respectively. The SPP policy, which places all the hash tree components contiguously, performs extremely well. This is not surprising since it imposes the smallest overhead, and the dynamic creation order to an extent matches the hash tree traversal order in the support counting phase. The custom memory allocation library in SPP also avoids polluting the cache with boundary tag information. We thus obtain a gain of 35-63% by using the simple scheme alone. For L-SPP, with a separate lock and counter region, we obtain slightly better results than SPP. As with all lock-region based schemes, L-SPP loses on locality, but with increasing data sets, the benefits of reducing false sharing outweigh that 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 GPP policy performs the best with increasing data size. It uses the hash tree traversal order to re-arrange the hash tree to maximize locality, resulting in significant performance gains. L-GPP outperforms GPP only for the biggest datasets. The best overall scheme is LCA-GPP, with upto a 20% gain over SPP, since it has a local counter array per processor that eliminates false sharing and lock synchronization completely, but at the same time it retains good locality.

Conclusions

In this paper, we proposed a set of policies for controlling the allocation and memory placement of dynamic data structures that arise in parallel association mining, to achieve the conflicting goals of maximizing locality and minimizing false sharing. Our experiments show that simple placement schemes can be quite effective, improving the execution time by a factor of two (upto 60% better than the base case). With larger datasets and at lower values of support, the more complex placement schemes yielded additional improvements (upto 20% better than simple scheme).

References

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

Agrawal, R., and Srikant, R. 1995. Mining sequential patterns. In *11th Intl. Conf. on Data Engg.*

Agrawal, R.; Mannila, H.; Srikant, R.; Toivonen, H.; and Verkamo, A. I. 1996. Fast discovery of association rules. In Fayyad, U., eds., *Advances in KDD*, 307–328. AAAI Press.

Anderson, J. M., and Lam, M. 1993. Global optimizations for parallelism and locality on scalable parallel machines. *ACM Conf. Programming Language Design and Implementation*.

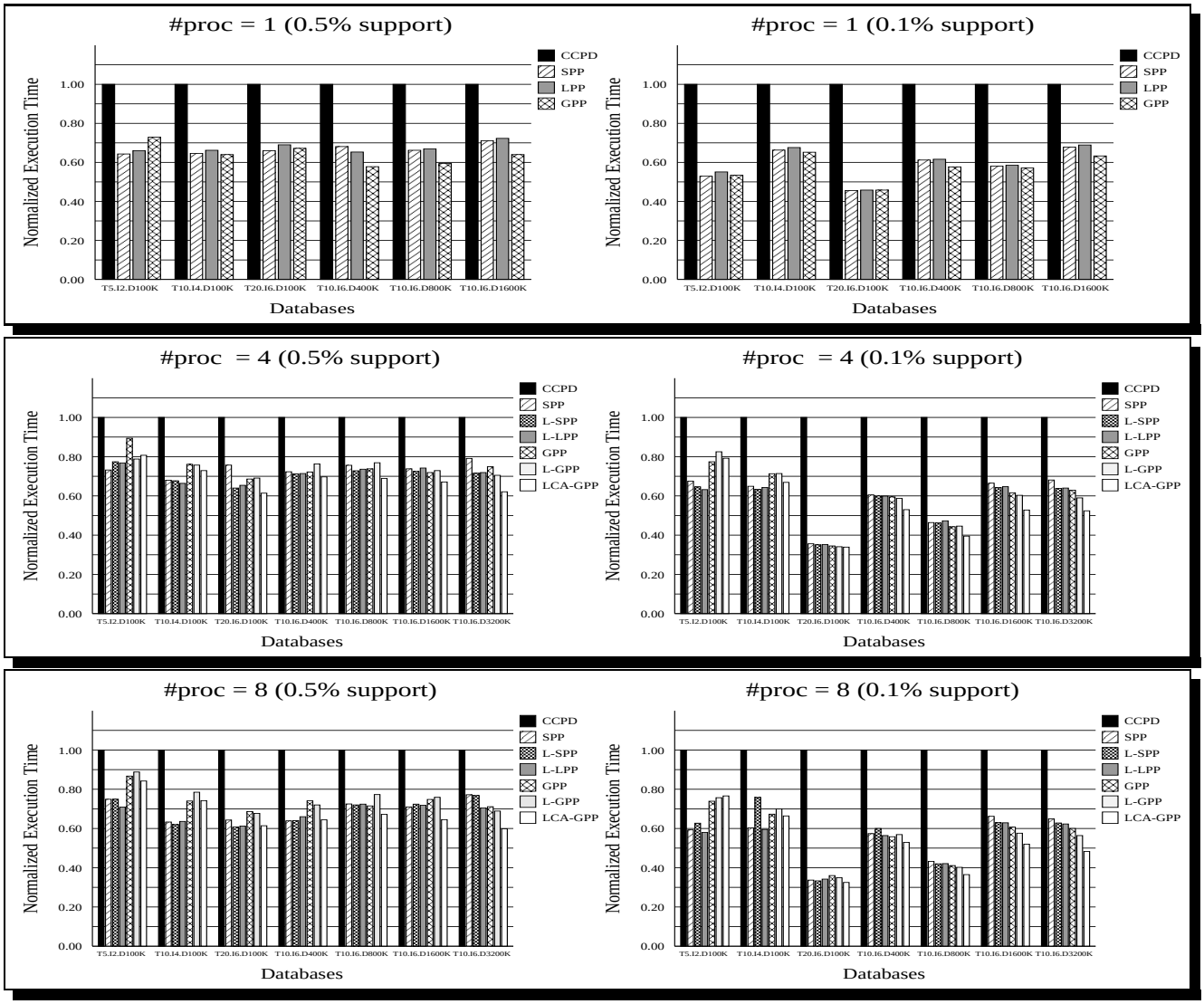

Figure 3: Memory Placement Policies: 1, 4, and 8 Processors

Anderson, J. M.; Amarsinghe, S. P.; and Lam, M. 1995. Data and computation transformations for multiprocessors. *ACM Symp. Principles and Practice of Parallel Programming*.

Bianchini, R., and LeBlanc, T. J. 1992. Software caching on cache-coherent multiprocessors. *4th Symp. Par. Dist. Processing.*

Carr, S.; McKinley, K. S.; and Tseng, C.-W. 1994. Compiler optimizations for improving data locality. *6th ACM ASPLOS Conf.*

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

Han, E.-H.; Karypis, G.; and Kumar, V. 1997. Scalable parallel data mining for association rules. In *ACM SIGMOD Conference*.

Hennessey, J., and Patterson, D. 1995. *Computer Architecture : A Quantitative Approach*. Morgan-Kaufmann Pub.

Luk, C.-K., and Mowry, T. C. 1996. Compiler-based prefetching for recursive data structures. *6th ACM ASPLOS Conference*.

Park, J. S.; Chen, M.; and Yu, P. S. 1995. Efficient parallel data mining for association rules. In *ACM Conf. Info. Know. Mgmt*.

Parthasarathy, S.; Zaki, M. J.; and Li, W. 1997. Custom memory placement for parallel data mining. U. of Rochester, TR 653.

Shintani, T., and Kitsuregawa, M. 1996. Hash based parallel algorithms for mining association rules. *4th Intl. Conf. Parallel and Distributed Info. Systems*.

Srikant, R., and Agrawal, R. 1995. Mining generalized association rules. In *21st VLDB Conference*.

Srikant, R., and Agrawal, R. 1996. Mining quantitative association rules in large relational tables. In *ACM SIGMOD Conference*.

Torrellas, J.; Lam, M. S.; and Hennessy, J. L. 1990. Shared data placement optimizations to reduce multiprocessor cache miss rates. *Intl. Conf. Parallel Processing* II:266–270.

Zaki, M. J.; Ogihara, M.; Parthasarathy, S.; and Li, W. 1996. Parallel data mining for association rules on shared-memory multiprocessors. In *Supercomputing'96*.

Zaki, M. J.; Parthasarathy, S.; Ogihara, M.; and Li, W. 1997. New parallel algorithms for fast discovery of association rules. *J. Data Mining and Knowledge Discovery* 1(4):343-373.