Parallel Data Mining for Association Rules on Shared-memory Multi-processors *

M. J. Zaki, M. Ogihara, S. Parthasarathy, and W. Li
{zaki,ogihara,srini,wei}@cs.rochester.edu

July 1996

Abstract

Data mining is an emerging research area, whose goal is to extract significant patterns or interesting rules from databases. High-level inference from large volumes of routine business data can provide valuable information to businesses, such as customer buying patterns, shelving criterion in supermarkets and stock trends. Many algorithms have been proposed for data mining of association rules. However, research so far has mainly focused on sequential algorithms.

In this paper we present parallel algorithms for data mining of association rules, and study the degree of parallelism, synchronization, and data locality issues on the SGI Power Challenge shared-memory multi-processor. We further present a set of optimizations for the sequential and parallel algorithms. Experiments show that a significant improvement of performance is achieved using our proposed optimizations. We also achieved good speed-up for the parallel algorithm, but we observe a need for parallel I/O techniques for further performance gains.

Keywords: Data Mining, Association Rules, Load Balancing, Hash Tree Balancing, Hashing, Shared-Memory Multi-processor

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

1 Introduction

With large volumes of routine business data having been collected, business organizations are increasingly turning to the extraction of useful information from such databases. Such high-level inference processes may provide information on customer buying patterns, shelving criterion in supermarkets, stock trends, etc. Data mining is an emerging research area, whose goal is to extract significant patterns or interesting rules from such databases. Data mining is in fact a broad area which combines research in machine learning, statistics and databases. It can be broadly classified into these categories [1]: Classification (Clustering) – finding rules that partition the database into finite, disjoint, and previously known (unknown) classes; Sequences – extracting commonly occurring sequences in ordered data; and Associations (a form of summarization) – find the set of most commonly occurring groupings of items. In this paper we will concentrate on data mining for association rules. Application domains for association rules range from decision support to telecommunications alarm diagnosis, and prediction. The prototypical application is the analysis of sales data.

The problem of mining association rules over *basket* data was introduced in [2, 3]. It can be formally stated as: Let $\mathcal{I} = \{i_1, i_2, \cdots, i_m\}$ 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, such that $T \subseteq \mathcal{I}$. An *association rule* is an expression $A \Rightarrow B$, where $A, B \subset \mathcal{I}$, are sets of items called *itemsets*, and $A \cap B = \emptyset$. Each itemset is said to have a *support* s if s% of the transactions in $\mathcal{D}$ contain the itemset. The association rule is said to have *confidence* c if c% of the transactions that contain A also contain B, i.e., $c = support(A \cup B)/support(A)$, i.e., the conditional probability that transactions contain the itemset B, given that they contain itemset A. Data mining of association rules from such databases consists of finding the set of all such rules which meet the user-specified minimum confidence and support values.

The data mining task for association rules can be broken into two steps. The first step consists of finding all large $itemsets$. The second step consists of forming implication rules with a user specified $confidence$ among the large itemsets found in the first step. Since this step is not compute intensive [5], henceforth we will focus only on the first step. The general structure of most algorithms for mining associations is that during the initial pass over the database the support for all single items (1-itemsets) is counted. The large 1-itemsets are used to generate candidate 2-itemsets. The database is scanned again to obtain occurrence counts for the candidates, and the large 2-itemsets are selected for the next pass. This iterative process is repeated for $k = 3, 4, \cdots$, until there are no more large k-itemsets to be found.

1.1 Related Work

Many algorithms for finding large itemsets have been proposed in the literature since the introduction of this problem in [2] (AIS algorithm).

In [9] a pass minimization approach was presented, which uses the idea that if an itemset belongs to the set of large $(k + e)$-itemsets, then it must contain $\binom{k+e}{k}$ k-itemsets. The *Apriori* algorithm [5] also uses the property that any subset of a large itemset must itself be large. These algorithms had performance superior to AIS and are also polynomial. The DHP algorithm [10] uses a hash table in pass k to do efficient pruning of $(k + 1)$-itemsets. The *Partition* algorithm [13] minimizes

I/O by scanning the database only twice. In the first pass it generates the set of all potentially large itemsets, and in the second pass the support for all these is measured. The above algorithms are all specialized black-box techniques which do not use any database operations. Algorithms using only general-purpose DBMS systems and relational algebra operations have also been proposed [7, 8]. The work closest to this from the machine learning literature is the KID3 algorithm presented in [12]. The main problem with their approach is that it may take exponential time in the worst case as opposed to the polynomial time algorithms presented in the above papers.

There has been very limited work in parallel implementations of association algorithms. In [11], a parallel implementation of the DHP algorithm [10] is presented. However only simulation results on a shared-nothing or distributed-memory machine like IBM SP2 were presented. Parallel implementations of the *Apriori* algorithm on the IBM SP2 were presented in [4]. There has been no study on shared-everything or shared-memory machines to-date.

1.2 Contribution

In this paper we present parallel implementations of the *Apriori* algorithm on the SGI Power Challenge shared-memory multi-processor. We study the degree of parallelism, synchronization, and data locality issues in parallelizing data mining applications for such architectures. We also present a set of optimizations for the sequential *Apriori* algorithm, and for the parallel algorithms as well.

The rest of the paper is organized as follows. In the next section we briefly describe the *Apriori* algorithm. Section 3 presents a discussion of the parallelization issues for each of the steps in the algorithm, while section 4 presents some effective optimizations for mining association rules. Section 5 presents our experimental results for the different optimization and the parallel performance. Finally we conclude in section 6.

2 The *Apriori* Algorithm

The naive method of finding large itemsets would be to generate all the 2^m subsets of the universe of m items, count their support by scanning the database, and output those meeting minimum support criterion. It is not hard to see that the naive method exhibits complexity exponential in m, and is quite impractical. *Apriori* follows the basic iterative structure discussed earlier. However the key observation used is that any subset of a large itemset must also be large. During each iteration of the algorithm only candidates found to be large in the previous iteration are used to generate a new candidate set to be counted during the current iteration. A pruning step eliminates any candidate which has a small subset. The algorithm terminates at step t, if there are no large t-itemsets. The general structure of the algorithm is given in figure 1, and a brief discussion of each step is given below (for details on its performance characteristics, we refer the reader to [5]). In the figure L_k denotes the set of large k-itemsets, and C_k the set of candidate k-itemsets.

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

$$\begin{aligned} C_k \;=\; \{x \;\mid\; & x[1:k-2] = A[1:k-2] = B[1:k-2],\ x[k-1] = A[k-1], \\ & x[k] = B[k-1],\ A[k-1] < B[k-1], \text{ where } A, B \in L_{k-1}\} \end{aligned}$$

```
L_1 = {large 1-itemsets };
for (k = 2; L_{k-1} ≠ ∅; k++)
   /*Candidate Itemsets generation*/
   C_k = Set of New Candidates;
   /*Support Counting*/
   for all transactions t ∈ D
      for all k-subsets s of t
         if (s ∈ C_k) s.count++;
   /*Large Itemsets generation*/
   L_k = {c ∈ C_k | c.count ≥ minimum support};
Set of all large itemsets = ⋃_k L_k;
```

Figure 1: The *Apriori* Algorithm

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 large. This pruning step eliminates any itemset at least one of whose subsets is not large. 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 $\mathcal{F}$. All the itemsets are stored in the leaves. 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. 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.

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. This is done by starting at the root and hashing on items 0 through $(n-k+1)$ of the transaction. If we reach depth d by hashing on item i then we hash on items i through $(n-k+1)+d$. This is done recursively, until we reach a leaf. At this point we increment the count of all itemsets in the leaf that are contained in the transaction (note: this is the reason for having a small threshold value).

For large itemsets generation, each itemset in C_k with minimum support is inserted into L_k, a sorted linked-list denoting the large k-itemsets. L_k is used in generating the candidates in the next iteration.

3 Parallel Data Mining: Design Issues

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. In our algorithm large itemsets generation is embedded in the support counting wherein as soon as a candidate has reached the minimum support it is added to the large set.

3.1 Candidate Itemsets Generation

3.1.1 Optimized Join and Pruning

Recall that in iteration k, C_k is generated by joining L_{k-1} with itself. The naive way of doing the join is to look at all $\binom{|L_{k-1}|}{2}$ combinations. However, since L_{k-1} is lexicographically sorted, we can partition the itemsets in L_{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 $\binom{|L_{k-1}|}{2}$ combinations.

While pruning a candidate we have to check if all k of its $(k-1)$-subsets are large. 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 ($\mathcal{F}$) : Having equivalence classes also allows us to accurately adapt the hash table size $\mathcal{F}$ for each iteration. For iteration k, and for a given *threshold* value $\mathcal{T}$, 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: $\mathcal{T}\mathcal{F}^k$. Since we can insert up to $\sum_{i=0}^{n} \binom{|S_i|}{2}$ itemsets, we get the expression: $\mathcal{T}\mathcal{F}^k \geq \sum_{i=0}^{n} \binom{|S_i|}{2}$. This can be solved to obtain:

$$\mathcal{F} \geq \left(\frac{\sum_{i=0}^{n} \binom{|S_i|}{2}}{\mathcal{T}} \right)^{1/k}$$

3.1.2 Computation Balancing

Let the number of processors $\mathcal{P} = 3$, $k = 2$, and $L_1 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. There is only 1 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 $\mathcal{P}$ processors.

Block partitioning : A simple block partition generates the assignment $\mathcal{A}_0 = \{0, 1, 2\}$, $\mathcal{A}_1 = \{3, 4, 5\}$ and $\mathcal{A}_2 = \{6, 7, 8, 9\}$, where $\mathcal{A}_p$ denotes the itemsets assigned to processor p. The resulting workload per processor is: $\mathcal{W}_0 = 9 + 8 + 7 = 24$, $\mathcal{W}_1 = 6 + 5 + 4 = 15$ and $\mathcal{W}_2 = 3 + 2 + 1 = 6$, where $\mathcal{W}_p = \sum_{i \in \mathcal{A}_p} w_i$. 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 $\mathcal{A}_0 = 0, 3, 6, 9$, $\mathcal{A}_1 = 1, 4, 7$ and $\mathcal{A}_2 = 2, 5, 8$. The work load is now given as

$\mathcal{W}_0 = 9 + 6 + 3 = 18$, $\mathcal{W}_1 = 8 + 5 + 2 = 15$ and $\mathcal{W}_2 = 7 + 4 + 1 = 12$. The load imbalance is much smaller, however it is still present.

Bitonic Partitioning (Single Equivalence Class): In [6] 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 $(2\mathcal{P} - i - 1)$ is a constant:

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

We can therefore assign itemsets i and $(n - i - 1)$ as one *unit* with uniform work $(2n - 2\mathcal{P}t - 1)$. If $n \bmod 2\mathcal{P} = 0$ then perfect balancing results. The case $n \bmod 2\mathcal{P} \neq 0$ is handled as described in [6].

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

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 work load 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 [14], the major difference being work loads 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 $\binom{n}{k}$ large 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 considered in Section 5. 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 large k-itemsets. If there aren't a sufficient number of large 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. This complication only arises at the interface of the leaves and internal nodes.

With this locking mechanism, each process can insert the itemsets in different parts of the hash tree in parallel. However, since we start with a hash tree with the root as a leaf, there can be a lot of initial contention to acquire the lock at the root. However, we did not find this to be a significant factor on 12 processors.

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. There are several ways of incrementing the count of itemsets in the common candidate hash tree.

Counter per Itemset: Let us assume that each itemset in the candidate hash tree has a single count field associated with it. Since the counts are common, more than one processor may try to access the count field and increment it. We thus need a locking mechanism to provide mutual exclusion among the processors while incrementing the count. This approach may cause contention and degrade the performance. However, since we are using only 12 processors, and the sharing is very fine-grained (at the itemset level), we found this approach to be the better than using private or separate counters [1].

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 work load 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

[1] For all the databases we looked at on our system the overhead of contention was within 4%, which leads us to conclude that contention is not a big problem. Other mechanisms like separate counters (to eliminate locking) and local counters (to eliminate false sharing) were studied but not shown to be beneficial [14].

$\binom{l_t}{k}$ subsets of the transaction are contained in C_k. Clearly the complexity of the work load 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 won't 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 work load for each transaction over all iterations, given as $(\sum_{k=1}^{T} \binom{l_i}{k})/T$. 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 (see section 3.2.1) for each itemset. Finally, the master process selects the large 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 large 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. For this reason we did not implement these parallelizations.

4 Optimizations

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 work load, 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 $\mathcal{F}$ for

the hash table. We label the n large 1-itemsets from 0 to $n-1$ in lexicographical order, and use $P = \mathcal{F}$ to derive the assignments $\mathcal{A}_0, \cdots, \mathcal{A}_{\mathcal{F}-1}$ for each processor. Each $\mathcal{A}_i$ is treated as an equivalence class. The hash function is based on these equivalence classes, which is simply given as, $h(i) = \mathcal{A}_i$, for $i = 0, \cdots, \mathcal{F}$. The equivalence classes are implemented via an indirection vector of length n. For example, let $L_1 = \{A, D, E, G, K, M, N, S, T, Z\}$. We first label these as $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Assume that the fan-out $\mathcal{F} = 3$. We thus obtain the 3 equivalence classes $\mathcal{A}_0 = \{0, 5, 6\}$, $\mathcal{A}_1 = \{1, 4, 7\}$, and $\mathcal{A}_2 = \{2, 3, 8, 9\}$, and the indirection vector is shown in table 1. Furthermore, this hash function is applied at all levels of the hash tree. Clearly, this scheme results in a balanced hash tree as compared to the simple $g(i) = i \bmod \mathcal{F}$ hash function (which corresponds to the interleaved partitioning scheme from section 3.1.1).

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

Table 1: Indirection Vector

Balancing $C_k (k > 2)$: Although items can be pruned for iteration $k \geq 3$, 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 $k \geq 1$ denote the iteration number, $\mathcal{I} = \{0, \ldots, d-1\}$ the set of items, $\mathcal{F}$ the fan-out of the hash table, $T = \{0, \ldots, \mathcal{F}-1\}$ the set of equivalence classes modulo $\mathcal{F}$, $\mathcal{T} = T^k$ the total number of leaves in C_k, and $\mathcal{G}$ the family of all size k ordered subsets of $\mathcal{I}$, i.e., the set of all k-itemsets that can be constructed from items in $\mathcal{I}$. Suppose $\frac{d}{2\mathcal{F}}$ is an integer and $\frac{d}{2\mathcal{F}}, \mathcal{F} \geq k$. Define the bitonic hash function $h : \mathcal{I} \to T$ by:*

$$h(i) = i \bmod \mathcal{F} \text{ if } 0 \leq (i \bmod 2\mathcal{F}) < \mathcal{F} \text{ and } 2\mathcal{F} - 1 - (i \bmod 2\mathcal{F}) \text{ otherwise,}$$

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

$$e^{-\frac{k^2}{d/\mathcal{F}}} \leq \frac{\|H^{-1}(B)\|}{\|\mathcal{G}\|/\|\mathcal{T}\|} \leq e^{\frac{k^2}{d/\mathcal{F}}}.$$

A proof of the above theorem can be found in [14]. We also obtain the same lower and upper bound for the interleaved hash function also. However, the two functions behave differently. Note that the average number of k-itemsets per leaf $\|\mathcal{G}\|/\|\mathcal{T}\|$ is $\binom{2w\mathcal{F}}{k}/\mathcal{F}^k \approx \frac{(2w)^k}{k!}$. 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, $1 \leq i \leq k-1$. Thus, there are $\mathcal{F}(\mathcal{F}-1)^{k-1}$ such leaves, and so, $(1-\mathcal{F}^{-1})^{k-1}$ fraction of the leaves have capacity close to $\alpha(w)$. Note also that clearly, $(1-\mathcal{F}^{-1})^{k-1}$ 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 $k \geq 3$, 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

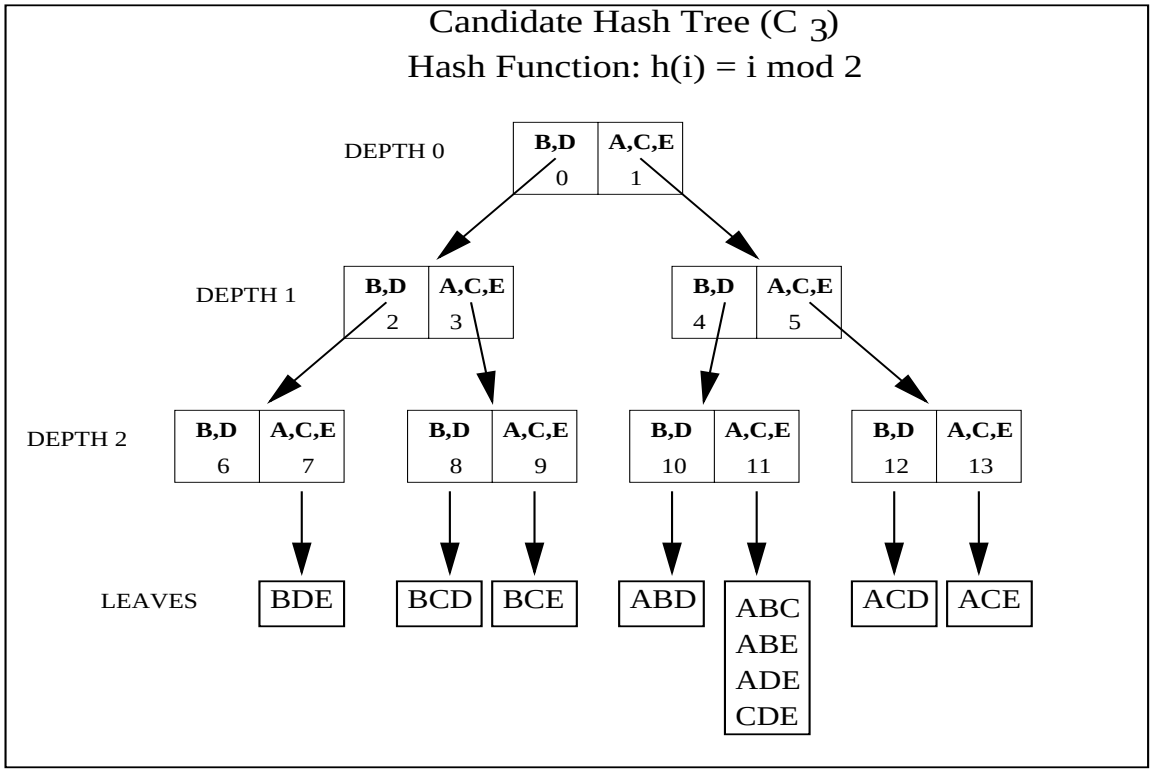

Figure 2: Candidate Hash Tree (C_3)

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 $\mathcal{F}$, for iteration k, we need additional memory of size $\mathcal{F}^k$ to store the flags. In the parallel implementation we have to keep a VISITED field for each processor, bringing the memory requirement to $\mathcal{P} \cdot \mathcal{F}^k$. This can still get very large, especially with increasing number of processors. In [14] we show a mechanism by which further reduces the memory requirement to only $k \cdot \mathcal{F}$. The approach in the parallel setting yields a total requirement of $k \cdot \mathcal{F} \cdot \mathcal{P}$.

5 Experimental Evaluation

Database	T	I	$\mathcal{D}$	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

Table 2: Database properties

5.1 Experimental Setup

All the experiments were performed on a 12-node SGI Power Challenge shared-memory multiprocessor. Each node is a MIPS processor running at 100MHz. There's a total of 256MB of main memory. The primary cache size is 16 KB (64 bytes cache line size), with different instruction and data caches, while the secondary cache is 1 MB (128 bytes cache line size). The databases are stored on an attached 2GB disk. All processors run IRIX 5.3, and data is obtained from the disk via an NFS file server.

We used different synthetic databases with size ranging form 3MB to 70MB [2], and are generated using the procedure described in [5]. These databases mimic the transactions in a retailing environment. Each transaction has a unique ID followed by a list of items bought in that transaction. The

[2]While results in this section are only shown for memory resident databases, the concepts and optimization are equally applicable for non memory resident databases. In non memory resident programs I/O becomes an important problem. Solutions to the I/O problem, can be applied in combination with the schemes presented in this paper. These solutions are part of future research.

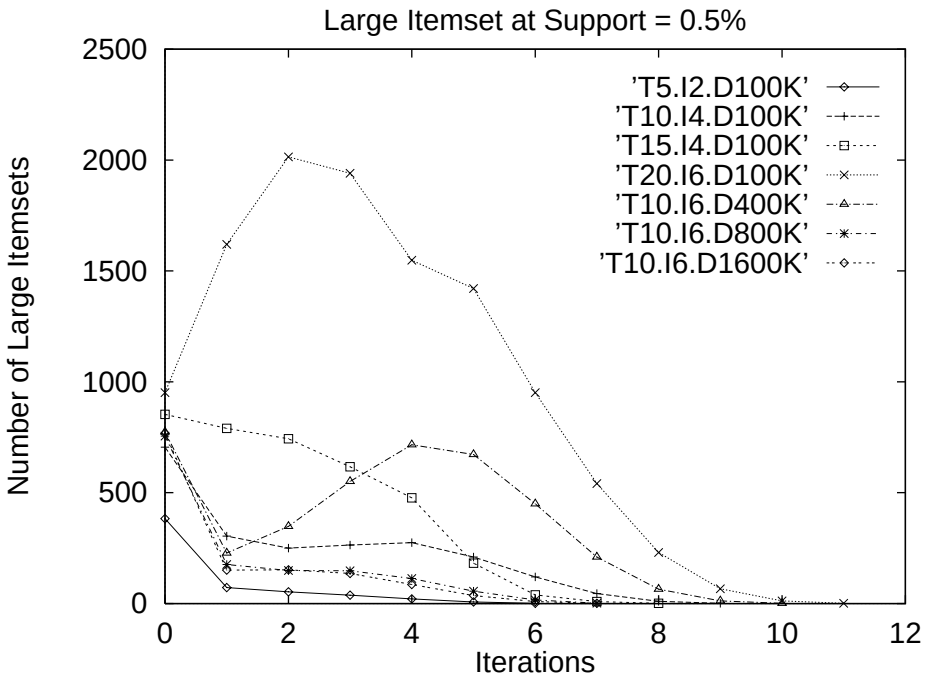

Figure 3: Large Itemsets per Iteration

data-mining provides information about the set of items generally bought together. Table 2 shows the databases used and their properties. The number of transactions is denoted as $|\mathcal{D}|$, average transaction size as $|T|$, and the average maximal potentially large itemset size as $|I|$. The number of maximal potentially large itemsets $|L| = 2000$, and the number of items $N = 1000$. We refer the reader to [5] for more detail on the database generation. All the experiments were performed with a minimum support value of 0.5%, and a leaf *threshold* of 2 (i.e., max of 2 itemsets per leaf). We note that the % improvements shown in all the experiments, except where indicated, do not take into account initial database reading time, since we specifically wanted to measure the effects of the optimizations on the computation. Figure 3 shows the number of iterations and the number of large itemsets found for different databases. 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 [3].

5.2 Aggregate Parallel Performance

Table 3 gives actual running times for the unoptimized sequential, and a naive parallelization of the base algorithm (Apriori) for 2,4 and 8 processors without any of the techniques descibed in sections 3 and 4. In this section all the graphs showing % improvements, are with respect to the data for one processor in table 3.

Figure 4 presents the speedups obtained on different databases and different processors for the CCPD parallelization. The results presented on CCPD use all the optimization discussed

[3]Recall that in the PCCD approach every processor has to read the entire database during each iteration. The resulting I/O costs on our system were too prohibitive for this method to be effective.

Database	1 proc	2 procs	4 procs	8 procs
T5.I2.D100K	20	17	12	10
T10.I4.D100K	96	70	51	39
T15.I4.D100K	236	168	111	78
T20.I6.D100K	513	360	238	166
T10.I6.D400K	372	261	165	105
T10.I6.D800K	637	435	267	163
T10.I6.D1600K	1272	860	529	307

Table 3: Naive Parallelization of Apriori (seconds)

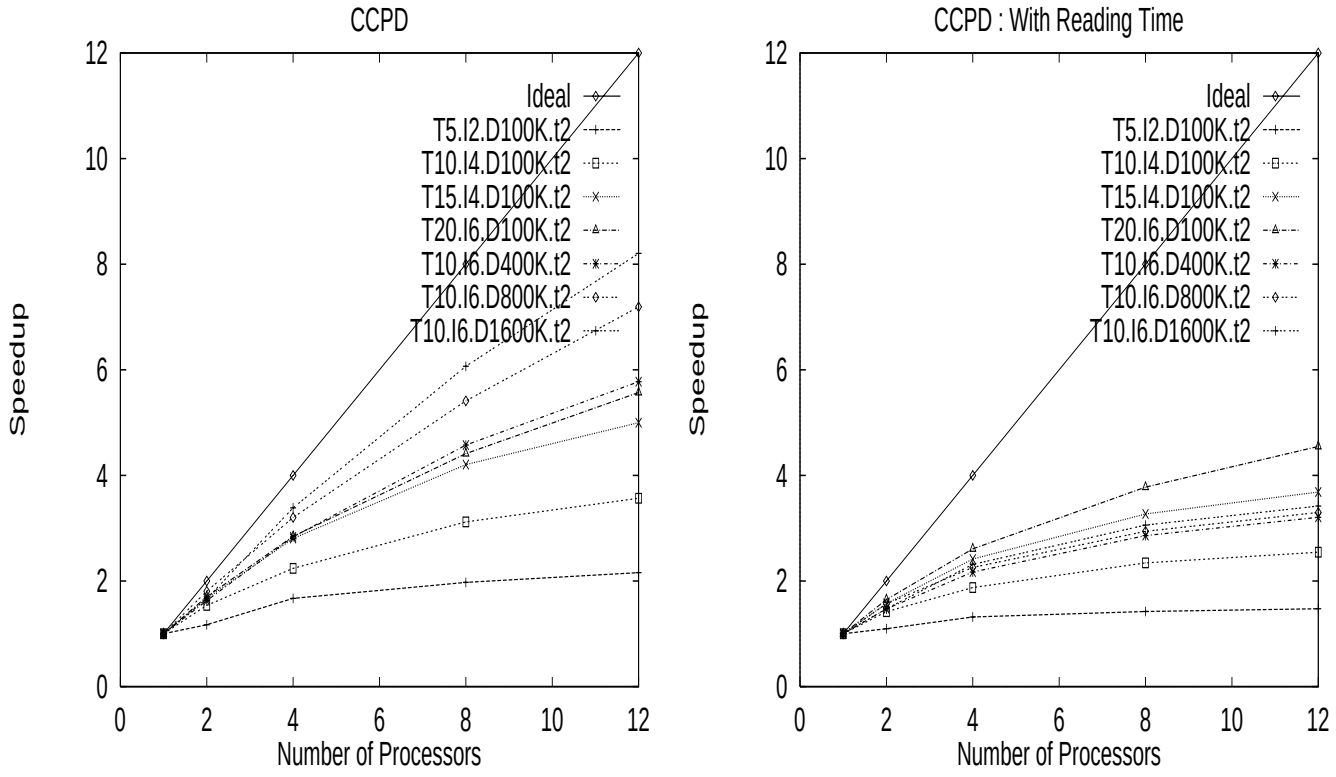

Figure 4: CCPD: Speed-up a) without reading time b) with reading time

Database	Reading Time	% of Total Time				
		$\mathcal{P}=1$	$\mathcal{P}=2$	$\mathcal{P}=4$	$\mathcal{P}=8$	$\mathcal{P}=12$
T5.I2.D100K	9.1s	39.9	43.8	52.6	56.8	59.0
T10.I4.D100K	13.7s	15.6	22.2	29.3	36.6	39.8
T15.I4.D100K	18.9s	8.9	14.0	21.6	29.2	32.8
T20.I6.D100K	24.1s	4.9	8.1	12.8	18.6	22.4
T10.I6.D400K	55.2s	16.8	24.7	36.4	48.0	53.8
T10.I6.D800K	109.0s	19.0	29.8	43.0	56.0	62.9
T10.I6.D1600K	222.0s	19.4	28.6	44.9	59.4	66.4

Table 4: Database Reading Time

in section 4, – computation balancing, hash tree balancing and short-circuited subset checking. The figure on the left presents the speed-up without taking the initial database reading time into account. We observe that as the number of transactions increase we get increasing speed-up, with a speed-up of more than 8 on 12 processors for the largest database (T10.I6.D1600K, with 1.6 million transactions). However, if we were to account for the database reading time, then we get speed-up of only 4 on 12 processors. The lack of linear speedup can be attributed to false and true sharing for the heap nodes when updating the subset counts and to some extent during the heap generation phase. Furthermore since variable length transactions are allowed, and the data is distributed along transaction boundaries, the workload is not be uniformly balanced. Other factors like bus contention, and i/o contention further reduce the speedup.

Table 4 shows the total time spent reading the database, and the percentage of total time this constitutes on different number of processors. The results indicate that on 12 processors up to 60% of the time can be spent just on I/O. This suggest a great need for parallel I/O techniques for effective parallelization of data mining applications since by its very nature data mining algorithms must operate on large amounts of data.

5.3 Computation and Hash Tree Balancing

Figure 5 shows the improvement in the performance obtained by applying the computation balancing optimization (discussed in section 3.1.2), and the hash tree balancing optimization (described in section 4.1). The figure shows the % improvement over a run on the same number of processors without any optimizations (see Table 3). 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 uni-processor case, as there is nothing to balance. However, it is very effective on multiple processors. We get an improvement of around 20% on 8 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 8 processors however it is not as

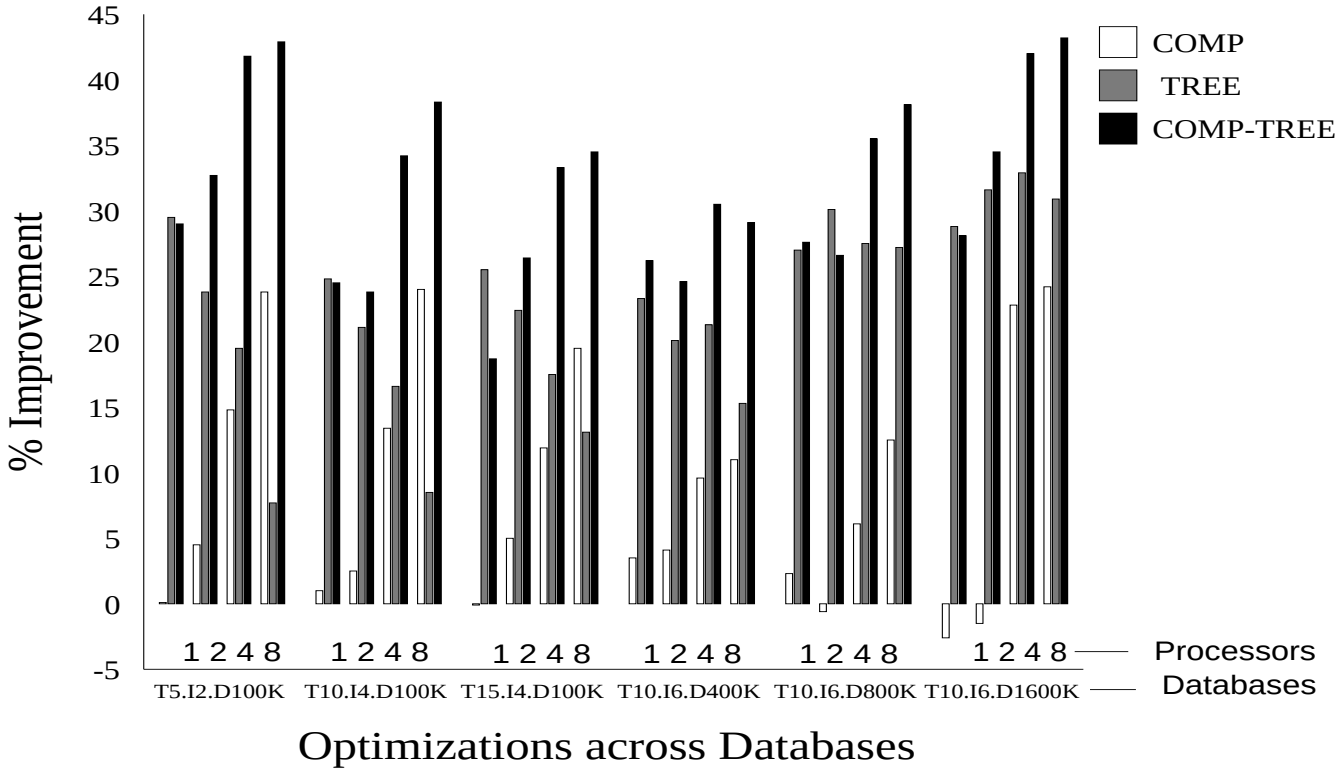

Figure 5: Effect of Computation and Hash Tree Balancing

good as the COMP optimization. The reason that the hash tree balancing is not sufficient to offset inherent load imbalance in the candidate generation in this case. The most effective approach is to apply both 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 1 processor only hash tree balancing is beneficial, since computation balancing only adds extra cost.

5.4 Short-circuited Subset Checking

Figure 6 shows the improvement due to the short-circuited subset checking optimization with respect to the unoptimized version (The unoptimized version is the Apriori algorithm due to Agrawal et al [5]). The results are presented for different number 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 figure 7. 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. It becomes clear that is an extremely effective

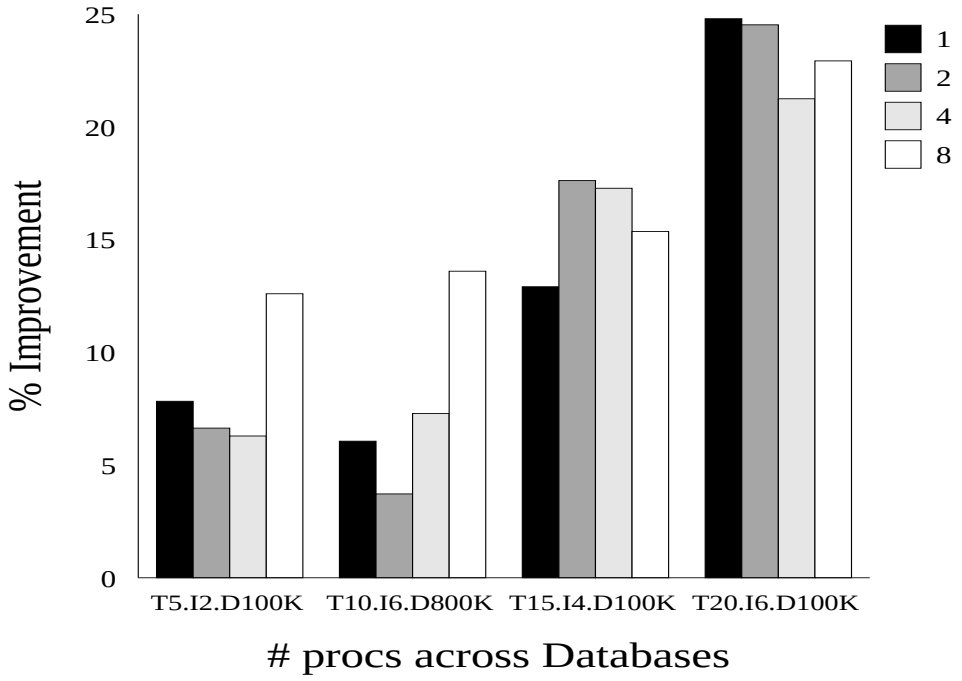

Figure 6: Effect of Short-circuited Subset Checking

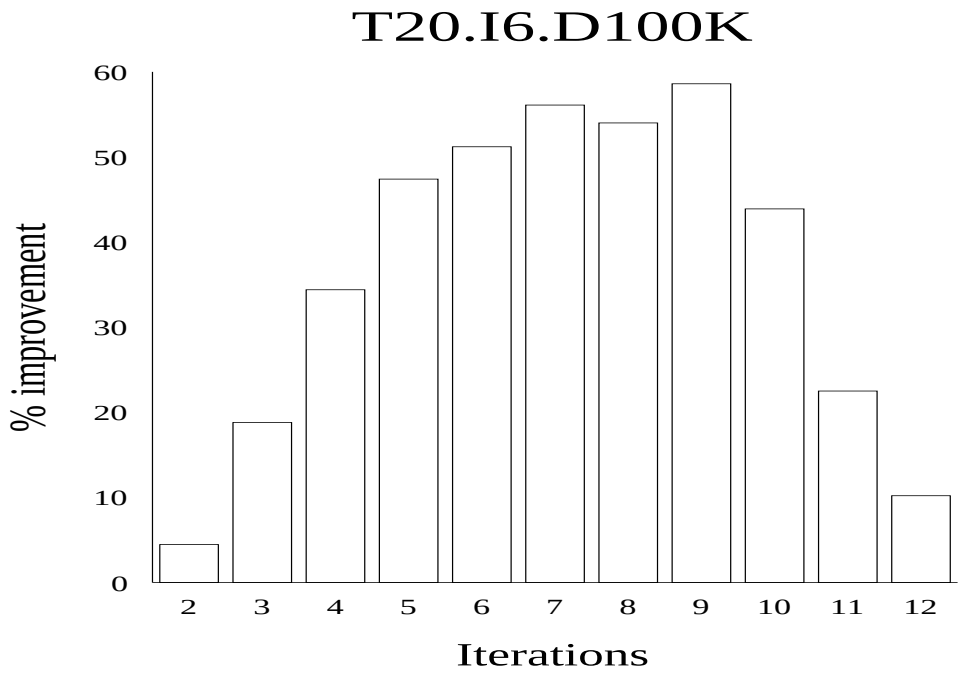

Figure 7: % Improvement per Iteration (# proc = 1)

optimization for larger transaction sizes, and in cases where there are large number of candidate k-itemsets.

6 Conclusions

In this paper, we presented a parallel implementation of the *Apriori* algorithm on the SGI Power Challenge shared memory multi-processor. We also 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 extremely effective for databases with large transaction sizes. Finally we reported the parallel performance of the algorithm. While we achieved good speed-up, we observed a need for parallel I/O techniques for further performance gains.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. In *IEEE Trans. on Knowledge and Data Engg.*, pages 5(6):914–925, 1993.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. ACM SIGMOD Intl. Conf. Management of Data*, May 1993.

[3] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. In U. F. et al, editor, *Advances in Knowledge Discovery and Data Mining*. MIT Press, 1996.

[4] R. Agrawal and J. Shafer. Parallel mining of association rules: design, implementation, and experience. Technical Report RJ10004, IBM Almaden Research Center, San Jose, CA 95120, Jan. 1996.

[5] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 20th VLDB Conf.*, Sept. 1994.

[6] M. Cierniak, W. Li, and M. J. Zaki. Loop scheduling for heterogeneity. In *4th IEEE Intl. Symposium on High-Performance Distributed Computing, also as URCS-TR 540, CS Dept., Univ. of Rochester*, Aug. 1995.

[7] M. Holsheimer, M. Kersten, H. Mannila, and H. Toivonen. A perspective on databases and data mining. In *1st Intl. Conf. Knowledge Discovery and Data Mining*, Aug. 1995.

[8] M. Houtsma and A. Swami. Set-oriented mining of association rules. In *RJ 9567*. IBM Almaden, Oct. 1993.

[9] H. Mannila, H. Toivonen, and I. Verkamo. Efficient algorithms for discovering association rules. In *AAAI Wkshp. Knowledge Discovery in Databases*, July 1994.

[10] J. S. Park, M. Chen, and P. S. Yu. An effective hash based algorithm for mining association rules. In *Proc. ACM SIGMOD Intl. Conf. Management of Data*, May 1995.

[11] J. S. Park, M. Chen, and P. S. Yu. Efficient parallel data mining for association rules. Technical Report RC20156, IBM T. J. Watson Research Center, Aug. 1995.

[12] G. Piatetsky-Shapiro. Discovery, presentation and analysis of strong rules. In G. P.-S. et al, editor, *KDD*. AAAI Press, 1991.

[13] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. 21st VLDB Conf.*, 1995.

[14] M. J. Zaki, M. Ogihara, S. Parthasarathy, and W. Li. Parallel data mining for association rules on shared-memory multi-processors. Technical Report 618, Department of Computer Science, University of Rochester, 618 1996.