Scalable Parallel Data Mining for Association Rules

Eui-Hong (Sam) Han, George Karypis, and Vipin Kumar

Abstract— **In this paper we propose two new parallel formulations of the *Apriori* algorithm that is used for computing association rules. These new formulations, *IDD* and *HD*, address the shortcomings of two previously proposed parallel formulations *CD* and *DD*. Unlike the *CD* algorithm, the *IDD* algorithm partitions the candidate set intelligently among processors to efficiently parallelize the step of building the hash tree. The *IDD* algorithm also eliminates the redundant work inherent in *DD*, and requires substantially smaller communication overhead than *DD*. But *IDD* suffers from the added cost due to communication of transactions among processors. *HD* is a hybrid algorithm that combines the advantages of *CD* and *DD*. Experimental results on a 128-processor Cray T3E show that *HD* scales just as well as the *CD* algorithm with respect to the number of transactions, and scales as well as *IDD* with respect to increasing candidate set size.**

Keywords— **Data mining, parallel processing, association rules, load balance, scalability.**

I. Introduction

One of the important problems in data mining [1] is discovering association rules from databases of transactions, where each transaction contains a set of items. The most time consuming operation in this discovery process is the computation of the frequencies of the occurrence of subsets of items, also called candidates, in the database of transactions. Since usually such transaction-based databases contain a large number of distinct items, the total number of candidates is prohibitively large. Hence, current association rule discovery techniques [2], [3], [4], [5] try to prune the search space by requiring a minimum level of support for candidates under consideration. Support is a measure of the number of occurrences of the candidates in database transactions. *Apriori* [2] is a recent state-of-the-art algorithm that aggressively prunes the set of potential candidates of size k by using the following observation: a candidate of size k can meet the minimum level of support only if all of its subsets also meet the minimum level of support. In the k^{th} iteration, this algorithm computes the occurrences of potential candidates of size k in each of the transactions. To do this task efficiently, the algorithm maintains all potential candidates of size k in a hash tree. This algorithm does not require the transactions to stay in main memory, but requires the hash trees to stay in main memory. If the entire hash tree cannot fit in the main memory, then the hash tree needs to be partitioned, and multiple passes over the transaction database need to be performed (one for each partition of the hash tree). Even with the highly effective pruning method of *Apriori*, the task of finding all association rules in many applications can require a lot of computation power that is available only in parallel computers.

Two parallel formulations of the *Apriori* algorithm were proposed in [6], *Count Distribution* (*CD*) and *Data Distribution* (*DD*). The *CD* algorithm scales linearly and has excellent speedup and sizeup behavior with respect to the number of transactions [6]. However, there are two problems with this algorithm. First, it does not parallelize the computation for building the hash tree. On a serial algorithm, this step takes relatively small amount of time. But on parallel computations, it can become a major bottleneck. Second, if the hash tree does not fit in the main memory, then the extra disk I/O for the multiple passes over the transaction database can be expensive on machines with slow I/O systems. Hence, the *CD* algorithm, like its sequential counterpart *Apriori*, is unscalable with respect to the increasing size of candidate set. The *DD* algorithm addresses these problems of the *CD* algorithm by partitioning the candidate set and assigning a partition to each processor in the system. However, this algorithm suffers from three types of inefficiency. First, the algorithm results in high communication overhead due to an inefficient scheme used for data movement. Second, the schedule for interactions among processors is such that it can cause processors to idle. Third, each transaction has to be processed against multiple hash trees causing redundant computation.

In this paper, we present two new parallel formulations of the *Apriori* algorithm for mining association rules. We first present *Intelligent Data Distribution* (*IDD*) algorithm that improves upon the *DD* algorithm by minimizing communication overhead and processor idling time, and by eliminating redundant computation. However, the static partitioning of the hash tree results in load imbalance that becomes severe for large number of processors. Furthermore, even with the optimized communication scheme, the communication overhead of *IDD* grows linearly with the number of transactions. Our second formulation, the *Hybrid Distribution* (*HD*) algorithm, combines the advantages of both the *CD* algorithm and the *IDD* algorithm by dynamically grouping processors and partitioning the candidate set accordingly to maintain good load balance. The experimental results on a Cray T3E parallel computer show that the *HD* algorithm scales very well and exploits the aggregate memory efficiently.

The rest of this paper is organized as follows. Section II provides an overview of the serial algorithm for mining association rules. Section III describes existing and proposed parallel algorithms. Section IV presents the performance analysis of the algorithms. Experimental results are shown in Section V. Section VI contains conclusions. A preliminary version of this paper appeared in [7].

II. Basic Concepts

Let T be the set of transactions where each transaction is a subset of the item-set I. Let C be a subset of I, then

TABLE I

Transactions from supermarket.

TID	Items
1	Bread, Coke, Milk
2	Beer, Bread
3	Beer, Coke, Diaper, Milk
4	Beer, Bread, Diaper, Milk
5	Coke, Diaper, Milk

we define the *support count* of C with respect to T to be:

$$\sigma(C) = |\{t | t \in T, C \subseteq t\}|.$$

Thus $\sigma(C)$ is the number of transactions that contain C. For example, consider a set of transactions from supermarket as shown in Table I. The items set I for these transactions is {Bread, Beer, Coke, Diaper, Milk}. The support count of {Diaper, Milk} is $\sigma(Diaper, Milk) = 3$, whereas $\sigma(Diaper, Milk, Beer) = 2$.

An *association rule* is an expression of the form $X \overset{s,\alpha}{\Longrightarrow} Y$, where $X \subseteq I$ and $Y \subseteq I$. The *support* s of the rule $X \overset{s,\alpha}{\Longrightarrow} Y$ is defined as $\sigma(X \cup Y)/|T|$, and the confidence α is defined as $\sigma(X \cup Y)/\sigma(X)$. For example, consider a rule {Diaper, Milk} $\Longrightarrow$ {Beer}, i.e. presence of diaper and milk in a transaction tends to indicate the presence of beer in the transaction. The support of this rule is $\sigma(Diaper, Milk, Beer)/5 = 40\%$. The confidence of this rule is $\sigma(Diaper, Milk, Beer)/\sigma(Diaper, Milk) = 66\%$. A rule that has a very high confidence (i.e., close to 1.0) is often very important, because it provides an accurate prediction on the association of the items in the rule. The support of a rule is also important, since it indicates how frequent the rule is in the transactions. Rules that have very small support are often uninteresting, since they do not describe significantly large populations. This is one of the reasons why most algorithms [2], [3], [4] disregard any rules that do not satisfy the minimum support condition specified by the user. This filtering due to the minimum required support is also critical in reducing the number of derived association rules to a manageable size. Note that the total number of possible rules is proportional to the number of subsets of the item-set I, which is $2^{|I|}$. Hence the filtering is absolutely necessary in most practical settings.

The task of discovering an association rule is to find all rules $X \overset{s,\alpha}{\Longrightarrow} Y$, such that s is greater than or equal to a given minimum support threshold and α is greater than or equal to a given minimum confidence threshold. The association rule discovery is composed of two steps. The first step is to discover all the frequent item-sets (candidate sets that have more support than the minimum support threshold specified). The second step is to generate association rules from these frequent item-sets. The computation of finding the frequent item-sets is much more expensive than finding the rules from these frequent item-sets. Hence in this paper, we only focus on the first step. The parallel implementation of the second step is straightforward and is discussed in [6].

F_1 = { frequent 1-item-sets } ;
for (k = 2; $F_{k-1} \neq \phi$; k++) {
    C_k = apriori_gen(F_{k-1})
    for all transactions $t \in T$ {
        subset(C_k, t)
    }
    $F_k = \{c \in C_k \mid c.count \geq minsup\}$
}
Answer = $\bigcup F_k$

Fig. 1. Apriori Algorithm

A number of sequential algorithms have been developed for discovering frequent item-sets [8], [2], [3]. Our parallel algorithms are based on the *Apriori* algorithm [2] that has smaller computational complexity compared to other algorithms. In the rest of this section, we briefly describe the *Apriori* algorithm. The reader should refer to [2] for further details.

The high level structure of the *Apriori* algorithm is given in Figure 1. The *Apriori* algorithm consists of a number of passes. Initially F_1 contains all the items (i.e., item set of size one) that satisfy the minimum support requirement. During pass k, the algorithm finds the set of frequent item-sets F_k of size k that satisfy the minimum support requirement. The algorithm terminates when F_k is empty. In each pass, the algorithm first generates C_k, the candidate item-sets of size k. Function *apriori_gen*(F_{k-1}) constructs C_k by extending frequent item-sets of size $k-1$. This ensures that all the subsets of size $k-1$ of a new candidate item-set are in F_{k-1}. Once the candidate item-sets are found, their frequencies are computed by counting how many transactions contain these candidate item-sets. Finally, F_k is generated by pruning C_k to eliminate item-sets with frequencies smaller than the minimum support. The union of the frequent item-sets, $\bigcup F_k$, is the frequent item-sets from which we generate association rules.

Computing the counts of the candidate item-sets is the most computationally expensive step of the algorithm. One naive way to compute these counts is to perform string-matching of each transaction against each candidate item-set. A faster way of performing this operation is to use a candidate hash tree in which the candidate item-sets are hashed [2]. Here we explain this via an example to facilitate the discussions of parallel algorithms and their analysis.

Figure 2 shows one example of the candidate hash tree with candidates of size 3. The internal nodes of the hash tree have hash tables that contain links to child nodes. The leaf nodes contain the candidate item-sets. A hash tree of candidate item-sets is constructed as follows. Initially, the hash tree contains only a root node, which is a leaf node containing no candidate item-set. When each candidate item-set is generated, the items in the set are stored in sorted order. Note that since C_1 and F_1 are created in

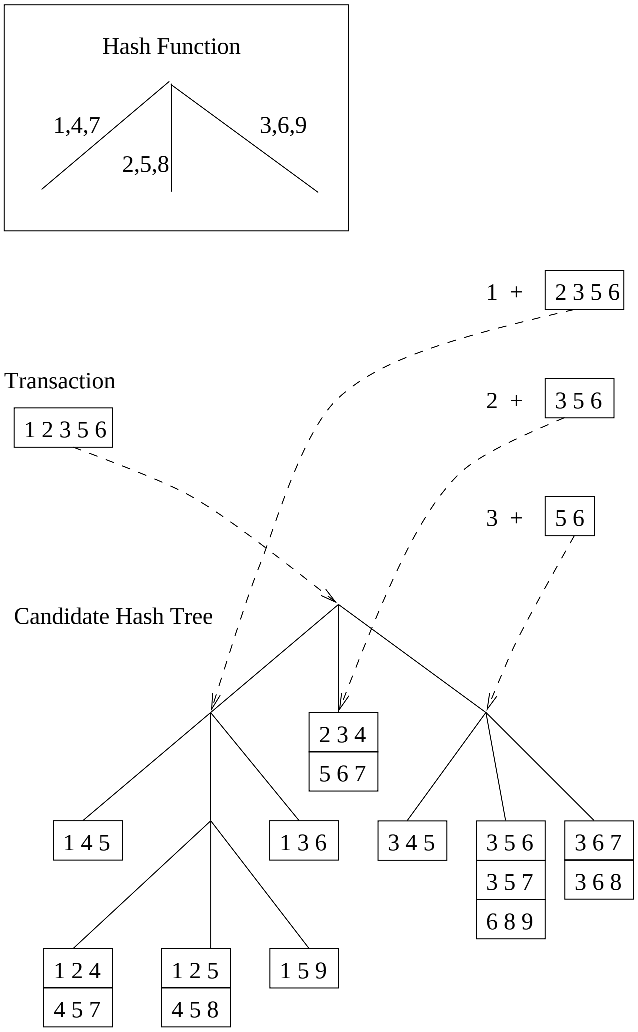

Fig. 2. Subset operation on the root of a candidate hash tree.

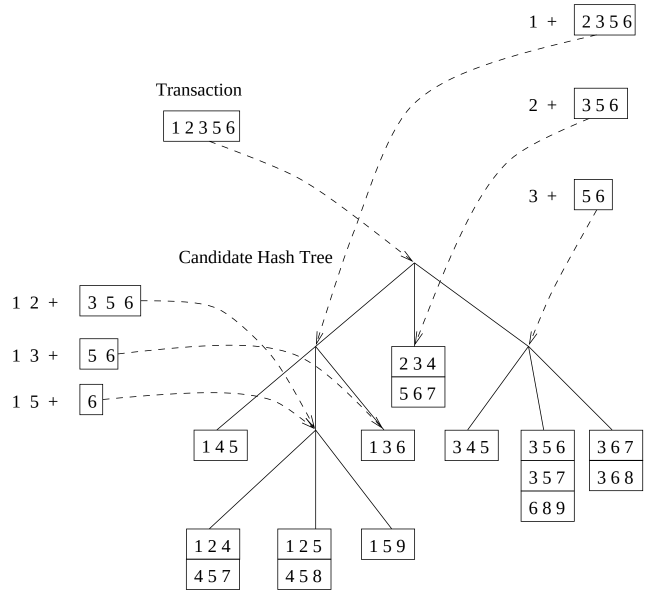

Fig. 3. Subset operation on the left most subtree of the root of a candidate hash tree.

sorted order, each candidate set is generated in sorted order without any need for explicit sorting. Each candidate item-set is inserted into the hash tree by hashing each successive item at the internal nodes and then following the links in the hash table. Once a leaf is reached, the candidate item-set is inserted at the leaf if the total number of candidate item-sets are less than the maximum allowed. If the total number of candidate item-sets at the leaf exceeds the maximum allowed and the depth of the leaf is less than k, the leaf node is converted into an internal node and child nodes are created for the new internal node. The candidate item-sets are distributed to the child nodes according to the hash values of the items. For example, the candidate item set $\{1\ 2\ 4\}$ is inserted by hashing item 1 at the root to reach the left child node of the root, hashing item 2 at that node to reach the middle child node, hashing item 3 to reach the left child node which is a leaf node.

The *subset* function traverses the hash tree from the root with every item in a transaction as a possible starting item of a candidate. In the next level of the tree, all the items of the transaction following the starting item are hashed. This is done recursively until a leaf is reached. At this time, all the candidates at the leaf are checked against the transaction and their counts are updated accordingly. Figure 2 shows the subset operation at the first level of the tree with transaction $\{1\ 2\ 3\ 5\ 6\}$. The item 1 is hashed to the left child node of the root and the following transaction $\{2\ 3\ 5\ 6\}$ is applied recursively to the left child node. The item 2 is hashed to the middle child node of the root and the whole transaction is checked against two candidate item-sets in the middle child node. Then item 3 is hashed to the right child node of the root and the following transaction $\{5\ 6\}$ is applied recursively to the right child node. Figure 3 shows the subset operation on the left child node of the root. Here the items 2 and 5 are hashed to the middle child node and the following transactions $\{3\ 5\ 6\}$ and $\{6\}$ respectively are applied recursively to the middle child node. The item 3 is hashed to the right child node and the remaining transaction $\{5\ 6\}$ is applied recursively to the right child node.

III. Parallel Algorithms

In this section, we will focus on the parallelization of the task that finds all frequent item-sets. We first discuss two parallel algorithms proposed in [6] to help motivate our parallel formulations. We also briefly discuss other parallel algorithms. In all our discussions, we assume that the transactions are evenly distributed among the processors.

A. Count Distribution Algorithm

In the *Count Distribution* (*CD*) algorithm proposed in [6], each processor computes how many times all the candidates appear in the locally stored transactions. This is done by building the entire hash tree that corresponds to all the candidates and then performing a single pass over the locally stored transactions to collect the counts. The global counts of the candidates are computed by summing these individual counts using a global reduction operation [9]. This algorithm is illustrated in Figure 4. Note that since each processor needs to build a hash tree for all the candidates, these hash trees are identical at each processor. Thus, excluding the global reduction, each processor in the *CD* algorithm executes the serial *Apriori* algorithm on the locally stored transactions.

This algorithm has been shown to scale linearly with the number of transactions [6]. This is because each processor can compute the counts independently of the other processors and needs to communicate with the other processors only once at the end of the computation step. However, this algorithm does not parallelize the computation of building the candidate hash tree. This step becomes a bottleneck with large number of processors. Furthermore, if the number of candidates is large, then the hash tree does not fit into the main memory. In this case, this algorithm has to partition the hash tree and compute the counts by scanning the database multiple times, once for each partition of the hash tree. The cost of extra database scanning can be expensive in the machines with slow I/O system. Note that the number of candidates increases if either the number of distinct items in the database increases or if the minimum support level of the association rules decreases. Thus the *CD* algorithm is effective for small number of distinct items and a high minimum support level.

B. Data Distribution Algorithm

The *Data Distribution* (*DD*) algorithm [6] addresses the memory problem of the *CD* algorithm by partitioning the candidate item-sets among the processors. This partitioning is done in a round robin fashion. Each processor is responsible for computing the counts of its locally stored subset of the candidate item-sets for all the transactions in the database. In order to do that, each processor needs to scan the portions of the transactions assigned to the other processors as well as its locally stored portion of the transactions. In the *DD* algorithm, this is done by having each processor receive the portions of the transactions stored in the other processors as follows. Each processor allocates P buffers (each one page long and one for each processor). At processor P_i, the i^{th} buffer is used to store transactions from the locally stored database and the remaining buffers are used to store transactions from the other processors. Now each processor P_i checks the P buffers to see which one contains data. Let l be this buffer (ties are broken in favor of buffers of other processors and ties among buffers of other processors are broken arbitrarily). The processor processes the transactions in this buffer and updates the counts of its own candidate subset. If this buffer corresponds to the buffer that stores local transactions (i.e., $l = i$), then it is sent to all the other processors (via asynchronous sends), and a new page is read from the local database. If this buffer corresponds to a buffer that stores transactions from another processor (i.e., $l \neq i$), then it is cleared and this buffer is marked available for next asynchronous receive from any other processors. This continues until every processor has processed all the transactions. Having computed the counts of its candidate item-sets, each processor finds the frequent item-sets from its candidate item-set and these frequent item-sets are sent to every other processor using an all-to-all broadcast operation [9]. Figure 5 shows the high level operations of the algorithm. Note that each processor has a different set of candidates in the candidate hash tree.

This algorithm exploits the total available memory better than *CD*, as it partitions the candidate set among processors. As the number of processors increases, the number of candidates that the algorithm can handle also increases. However, as reported in [6], the performance of this algorithm is significantly worse than the *CD* algorithm. The run time of this algorithm is 10 to 20 times more than that of the *CD* algorithm on 16 processors [6]. The problem lies with the communication pattern of the algorithm and the redundant work that is performed in processing all the transactions.

The communication pattern of this algorithm causes three problems. First, during each pass of the algorithm each processor sends to all the other processors the portion of the database that resides locally. In particular, each processor reads the locally stored portion of the database one page at a time and sends it to all the other processors by issuing $P-1$ send operations. Similarly, each processor issues a receive operation from each other processor in order to receive these pages. If the interconnection network of the underlying parallel computer is fully connected (i.e., there is a direct link between all pairs of processors) and each processor can receive data on all incoming links simultaneously, then this communication pattern will lead to a very good performance. In particular, if $O(N/P)$ is the size of the database assigned locally to each processor, the amount of time spent in the communication will be $O(N/P)$. However, even on the parallel computer with fully connected network, if each processor can receive data from (or send data to) only one other processor at a time, then the communication will be $O(N)$. On all realistic parallel computers, the processors are connected via a sparser networks (such as 2D, 3D or hypercube) and a processor can receive data from (or send data to) only one other processor at a time. On such machines, this communication pattern will take significantly more than $O(N)$ time because of contention within the network.

Second, in architectures without asynchronous communication support and with finite number of communication buffers in each processor, the proposed all-to-all communication scheme causes processors to idle. For instance, consider the case when one processor finishes its operation on local data and sends the buffer to all other processors. Now if the communication buffer of any receiving processors is full and the outgoing communication buffers are full, then the send operation is blocked.

Third, if we look at the size of the candidate sets as a function of the number of passes of the algorithm, we see that in the first few passes, the size of the candidate sets increases and after that it decreases. In particular, during the last several passes of the algorithm, there are only a

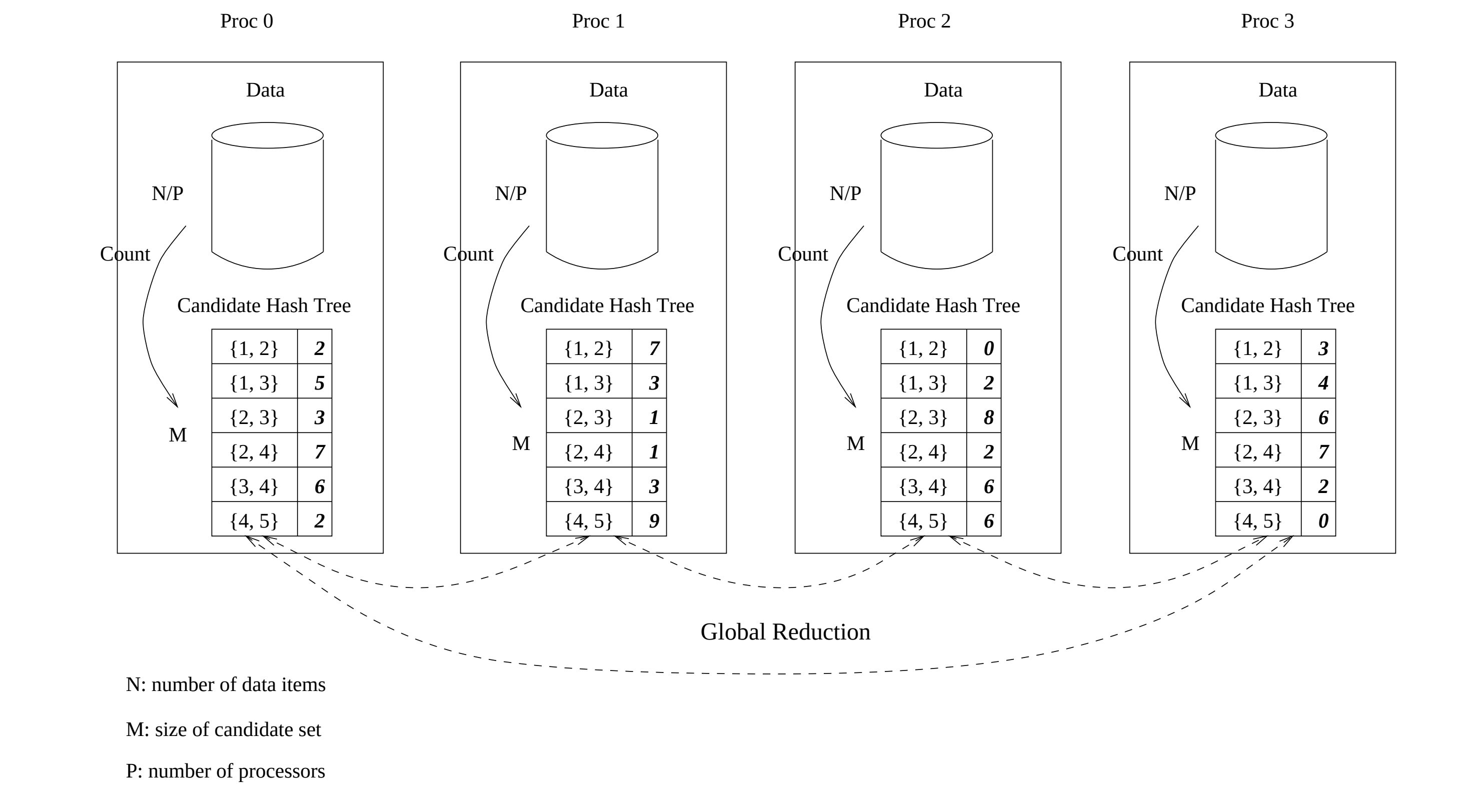

Fig. 4. Count Distribution (CD) Algorithm

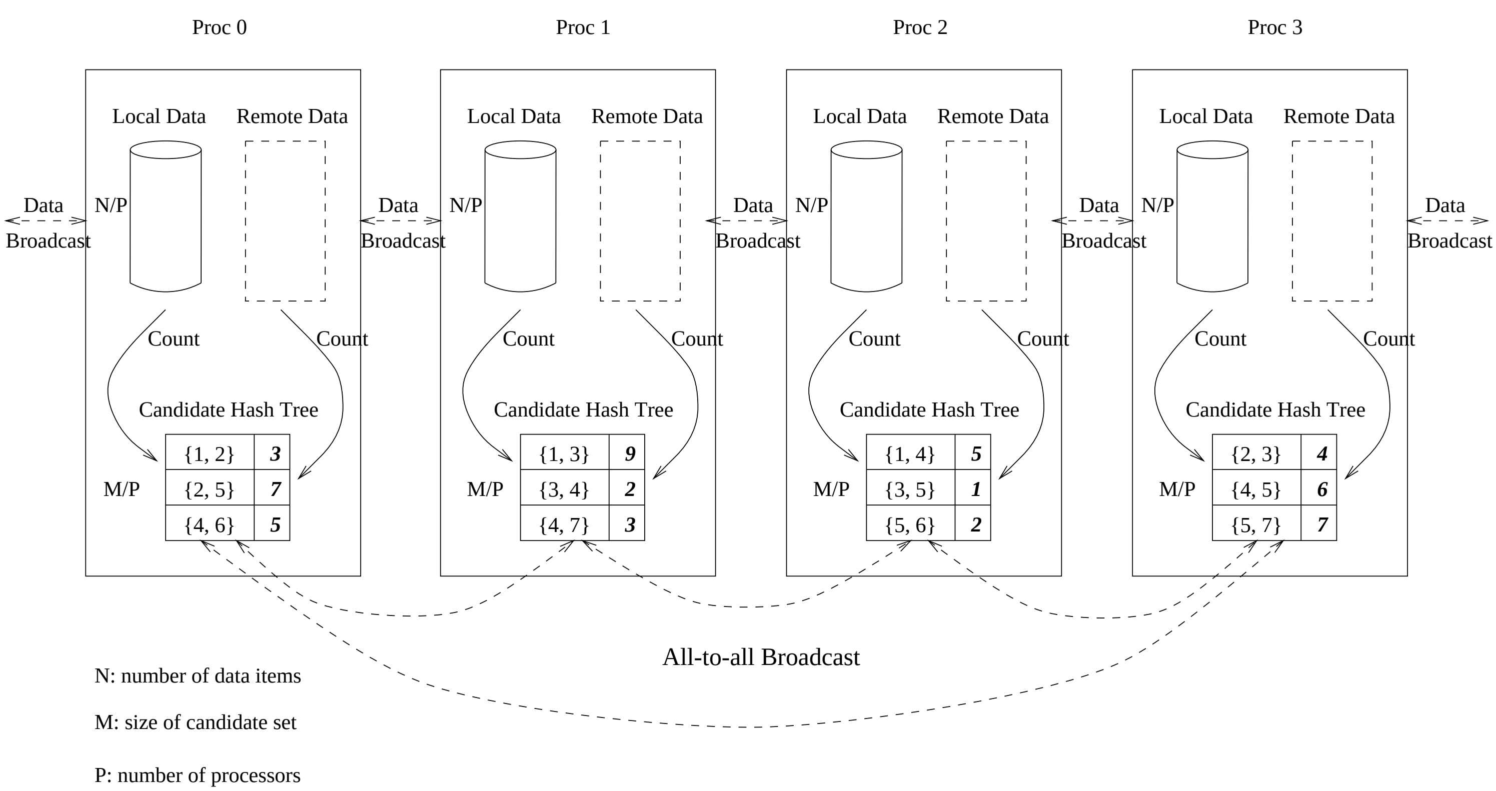

Fig. 5. Data Distribution (DD) Algorithm

small number of items in the candidate sets. However, each processor in the *DD* algorithm still sends the locally stored portions of the database to all the other processors. Thus, even though the computation decreases, the amount of communication remains the same.

The redundant work is introduced due to the fact that every processor has to process every single transaction in the database. In *CD* (see Figure 4), only N/P transactions go through each hash tree of M candidates, whereas in *DD* (see Figure 5), all N transactions have to go through each hash tree of M/P candidates. Although, the number of candidates stored at each processor has been reduced by a factor of P, the amount of computation performed for each transaction has not been proportionally reduced. If the amount of work required for each transaction to be checked against the hash tree of M/P candidates is $1/P$ of that of the hash tree of M candidates, then there is no extra work. As discussed in Section IV, in general, the amount of work per transaction will go down by a factor much smaller than P.

C. Intelligent Data Distribution Algorithm

We developed the *Intelligent Data Distribution* (*IDD*) algorithm that solves the problems of the *DD* algorithm discussed in Section III-B. In *IDD*, the locally stored portions of the database are sent to all the other processors by using a ring-based all-to-all broadcast described in [9]. This operation does not suffer from the contention problems of the *DD* algorithm and it takes $O(N)$ time on any parallel architecture that can be embedded in a ring. Figure 6 shows the pseudo code for this data movement operation. In our algorithm, the processors form a logical ring and each processor determines its right and left neighboring processors. Each processor has one send buffer (SBuf) and one receive buffer (RBuf). Initially, the SBuf is filled with one block of local data. Then each processor initiates an asynchronous send operation to the right neighboring processor with SBuf and an asynchronous receive operation to the left neighboring processor with RBuf. While these asynchronous operations are proceeding, each processor processes the transactions in SBuf and collects the counts of the candidates assigned to the processor. After this operation, each processor waits until these asynchronous operations complete. Then the roles of SBuf and RBuf are switched and the above operations continue for $P-1$ times. Compared to *DD*, where all the processors send data to all other processors, we perform only a point-to-point communication between neighbors, thus eliminating any communication contention. Furthermore, if the time to process a buffer does not vary much, then there is little time lost in idling.

In order to eliminate the redundant work due to the partitioning of the candidate item-sets, we must find a fast way to check whether a given transaction can potentially contain any of the candidates stored at each processor. This cannot be done by partitioning C_k in a round-robin fashion. However, if we partition C_k among processors in such a way that each processor gets item-sets that begin only with a subset of all possible items, then we can check the items of a transaction against this subset to determine if the hash tree contains candidates starting with these items. We traverse the hash tree with only the items in the transaction that belong to this subset. Thus, we solve the redundant work problem of *DD* by the intelligent partitioning of C_k.

```
while (!done) {
    FillBuffer(fd, SBuf);
    for (k = 0; k < P-1; ++k) {
        /* send/receive data in non-blocking pipeline */
        MPI_Irecv(RBuf, left);
        MPI_Isend(SBuf, right);

        /* process transactions in SBuf and update hash tree */
        Subset(HTree, SBuf);

        MPI_Waitall();

        /* swap two buffers */
        tmp = SBuf;
        SBuf = RBuf;
        RBuf = tmp;
    }
    /* process transactions in SBuf and update hash tree */
    Subset(HTree, SBuf);
}
```

Fig. 6. Pseudo Code for Data Movements

Figure 7 shows the high level picture of the algorithm. In this example, Processor 0 has all the candidates starting with items 1 and 7, Processor 1 has all the candidates starting with 2 and 5, and so on. Each processor keeps the first items of the candidates it has in a bit-map. In the *Apriori* algorithm, at the root level of hash tree, every item in a transaction is hashed and checked against the hash tree. However, in our algorithm, at the root level, each processor filters every item of the transaction by checking against the bit-map to see if the processor contains candidates starting with that item of the transaction. If the processor does not contain the candidates starting with that item, the processing steps involved with that item as the first item in the candidate can be skipped. This reduces the amount of transaction data that has to go through the hash tree; thus, reducing the computation. For example, let $\{1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\}$ be a transaction that processor 0 is processing in the *subset* function discussed in Section II. At the top level of the hash tree, processor 0 will only proceed with items 1 and 7 (i.e., 1 + 2 3 4 5 6 7 8 and 7 + 8). When the page containing this transaction is shifted to processor 1, this processor will only process items starting with 2 and 5 (i.e., 2 + 3 4 5 6 7 8 and 5 + 6 7 8). Figure 8 shows how this scheme works when a processor contains only those candidate item-sets that start with 1, 3 and 5. Thus for each transaction in the database, our approach partitions the amount of work to be performed among processors, thus eliminating most of the redundant work of *DD*. Note that both the judicious partitioning of the hash tree (indirectly caused by the partitioning of candidate item-set) and the filtering step are required to eliminate this redundant work.

The intelligent partitioning of the candidate set used in *IDD* requires our algorithm to have a good load balancing.

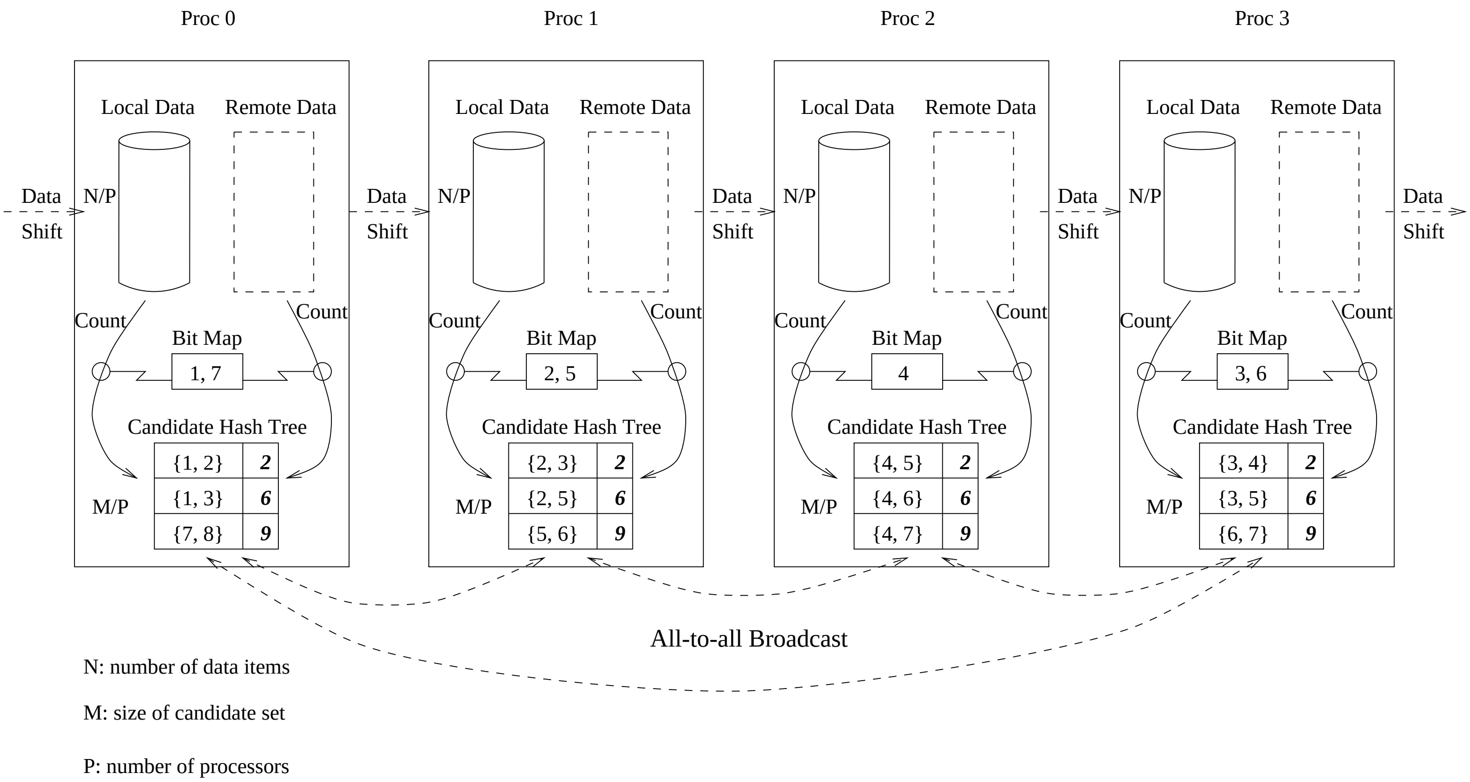

Fig. 7. Intelligent Data Distribution (IDD) Algorithm

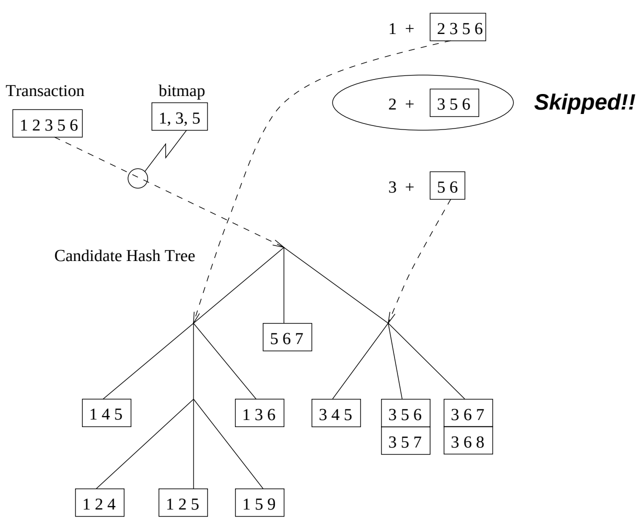

Fig. 8. Subset operation on the root of a candidate hash tree in IDD.

One of the criteria of a good partitioning involved here is to have an equal number of candidates in all the processors. This gives about the same size hash tree in all the processors and thus provides good load balancing among processors. Note that in the *DD* algorithm, this was accomplished by distributing candidates in a round robin fashion. A naive method for assigning candidates to processors can lead to a significant load imbalance. For instance, consider a database with 100 distinct items numbered from 1 to 100 and that the database transactions have more data items numbered with 1 to 50. If we partition the candidates between two processors and assign all the candidates starting with items 1 to 50 to processor P_0 and candidates starting with items 51 to 100 to processor P_1, then there would be more work for processor P_0.

To achieve an equal distribution of the candidate itemsets, we use a partitioning algorithm that is based on bin-packing [10]. For each item, we first compute the number of candidate item-sets starting with this particular item. Note that at this time we do not actually store the candidate item-sets, but just store the number of candidate item-sets starting with each item. We then use a bin-packing algorithm to partition these items in P buckets such that the sum of numbers of the candidate item-sets starting with these items in each bucket are roughly equal. Once the location of each candidate item-set is determined, then each processor locally regenerates and stores candidate item-sets that are assigned to this processor. Note that bin-packing is used per pass of the algorithm and the amount of time spent on bin-packing is minor compared to the overall runtime. Figure 7 shows the partitioned candidate hash tree and its corresponding bitmaps in each processor.

Note that this scheme will not be able to achieve an equal distribution of candidates if there are too many candidate itemsets starting with the same item. For example, if there are more than M/P candidates starting with the same item, then one processor containing candidates starting with this item will have more than M/P candidates even if no other candidates are assigned to it. This problem gets more serious with increasing P. One way of handling this problem is to partition candidate item sets based on more than the first items of the candidate item sets. In this approach, whenever the number of candidates starting with one particular item is greater than the threshold, this item set is further partitioned using the second item of the candidate item sets.

Note that the equal assignment of candidates to the processors does not guarantee the perfect load balance among processors. This is because the cost of traversal and checking at the leaf node are determined not only by the size and shape of the candidate hash tree, but also by the actual items in the transactions. However, in our experiments, we have observed a reasonably good correlation between the size of candidate sets and the amount of work done by each processor. For example, with 4 processors, we were able to obtain the the load imbalance of 1.3% in terms of the number of candidate sets, and this translated into 5.4% load imbalance in the actual computation time. With 8 processors, we had 2.3% load imbalance in the number of candidate sets, and this resulted in 9.4% load imbalance in the computation time. Since the effect of transactions on the work load cannot be easily estimated in advance, our scheme only ensures that each processor has roughly equal number of candidate itemsets in the local hash tree.

D. Hybrid Algorithm

The *IDD* algorithm exploits the total system memory by partitioning the candidate set among all processors. The average number of candidates assigned to each processor is M/P, where M is the number of total candidates. As more processors are used, the number of candidates assigned to each processor decreases. This has two implications. First, with fewer number of candidates per processor, it is much more difficult to balance the work. Second, the smaller number of candidates gives a smaller hash tree and less computation work per transaction. Eventually the amount of computation may become less than the communication involved. This would be more evident in the later passes of the algorithm as the hash tree size further decreases dramatically. This reduces overall efficiency of the parallel algorithm. This will be an even more serious problem in a system that cannot perform asynchronous communication.

The *Hybrid Distribution* (*HD*) algorithm addresses the above problem by combining the *CD* and the *IDD* algorithms in the following way. Consider a P-processor system in which the processors are split into G equal size groups, each containing P/G processors. In the *HD* algorithm, we execute the *CD* algorithm as if there were only P/G processors. That is, we partition the transactions of the database into P/G parts each of size $N/(P/G)$, and assign the task of computing the counts of the candidate set C_k for each subset of the transactions to each one of these groups of processors. Within each group, these counts are computed using the *IDD* algorithm. That is, the transactions and the candidate set C_k are partitioned among the processors of each group, so that each processor gets roughly $|C_k|/G$ candidate item-sets and N/P transactions. Now, each group of processors computes the counts using the *IDD* algorithm, and the overall counts are computing by performing a reduction operation among the P/G groups of processors.

The *HD* algorithm can be better visualized if we think of the processors as being arranged in a two dimensional grid of G rows and P/G columns. The transactions are partitioned equally among the P processors. The candidate set C_k is partitioned among the processors of each column of this grid. This partitioning of C_k is identical for each column of processors; i.e., the processors along each row of the grid get the same subset of C_k. Figure 9 illustrates the *HD* algorithm for a 3×4 grid of processors. In this example, the *HD* algorithm executes the *CD* algorithm as if there were only 4 processors, where the 4 processors correspond to the 4 processor columns. That is, the database transactions are partitioned in 4 parts, and each one of these 4 hypothetical processors computes the local counts of all the candidate item-sets. Then the global counts can be computed by performing the global reduction operation discussed in Section III-A. However, since each one of these hypothetical processors is made up of 3 processors, the computation of local counts of the candidate item-sets in a hypothetical processor requires the computation of the counts of the candidate item-sets on the database transactions sitting on the 3 processors. This operation is performed by executing the *IDD* algorithm within each of 4 hypothetical processors. This is shown in the step 1 of Figure 9. Note that processors in the same row have exactly the same candidates, and candidate sets along the each column partition the total candidate set. At the end of this operation, each processor has complete count of its local candidates for all the transactions located in the processors of the same column (i.e., of a hypothetical processor). Now a reduction operation is performed along the rows such that all processors in each row have the sum of the counts for the candidates in the same row. At this point, the count associated with each candidate item-set corresponds to the entire database of transactions. Now each processor finds frequent item-sets by dropping all those candidate item-sets whose frequency is less than the threshold for minimum support. These candidate item-sets are shown as shaded in Figure 9(b). In the next step, each processor performs all-to-all broadcast operation along the columns of the processor mesh. At this point, all the processors have the frequent sets and are ready to proceed to the next pass.

The *HD* algorithm determines the configuration of the processor grid dynamically. In particular, the *HD* algorithm partitions the candidate set into a big enough section and assign a group of processors to each partition. Let m be a user specified threshold. If the total number of candidates M is less than m, then the *HD* algorithm makes G equal to 1, which means that the *CD* algorithm is run on all the processors. Otherwise G is set to $\lceil M/m \rceil$. Table II shows how the *HD* algorithm chose the processor configuration based on the number of candidates at each pass with 64 processors and $m = 50K$.

The *HD* algorithm inherits all the good features of the *IDD* algorithm. It also provides good load balance and enough computation work by maintaining minimum number of candidates per processor. At the same time, the

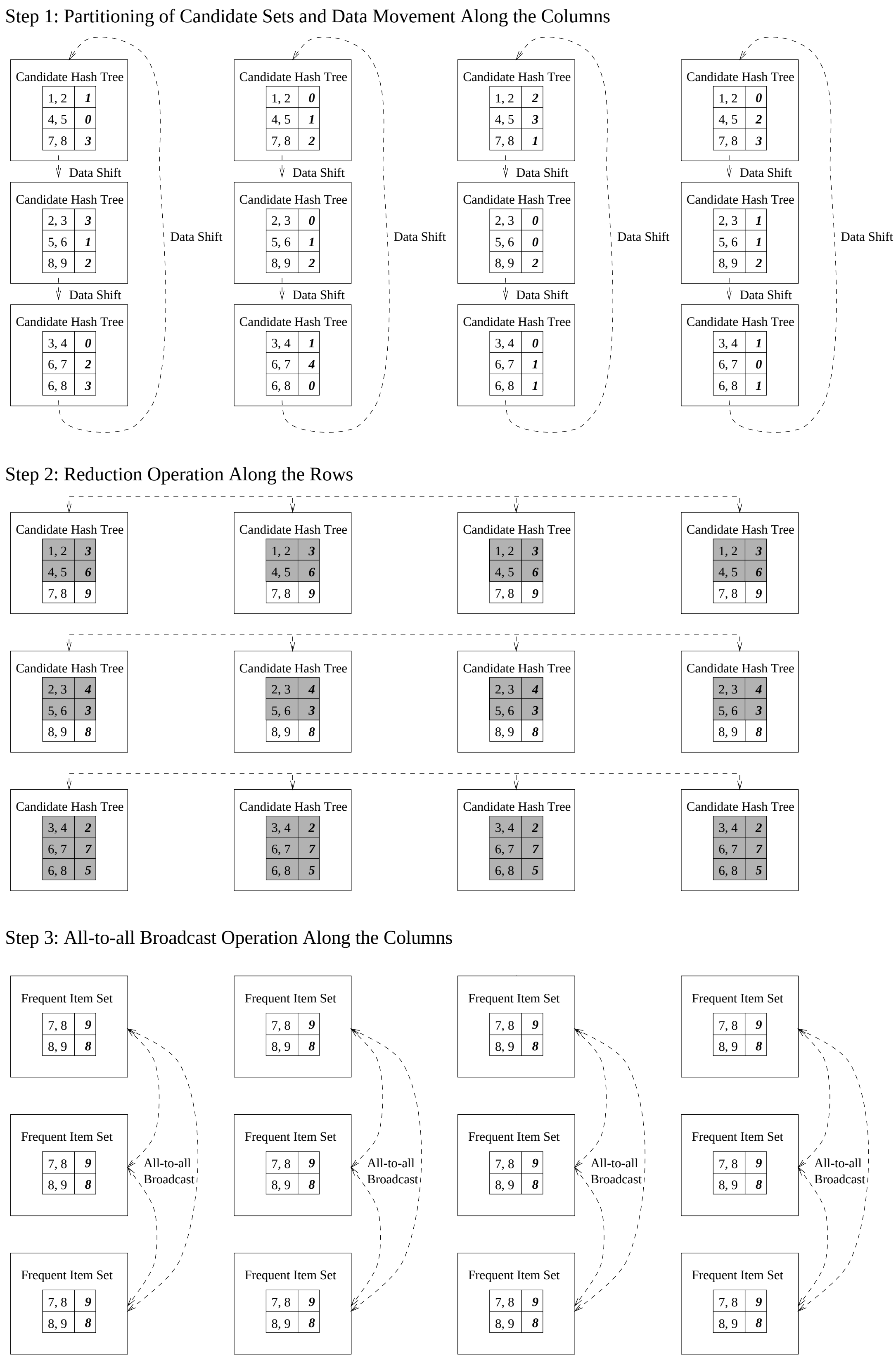

Fig. 9. Hybrid Distribution (HD) Algorithm in 3×4 Processor Mesh ($G = 3, P = 12$)

TABLE II

Processor configuration and number of candidates of the *HD* algorithm with 64 processors and with $m = 50K$ at each pass. Note that 64×1 configuration is the same as the *IDD* algorithm and 1×64 is the same as the *CD* algorithm. The total number of pass was 13 and all passes after 6 had 1×64 configuration.

Pass	2	3	4	5	6	7
Configuration	8×8	64×1	4×16	2×32	2×32	1×64
No of Cand.	351K	4348K	115K	76K	56K	34K

amount of data movement in this algorithm has been cut down to $1/G$ of the *IDD*.

E. Other Parallel Algorithms

In addition to *CD* and *DD*, four parallel algorithms (*NPA*, *SPA*, *HPA* and *HPA-ELD*) for mining association rules were proposed in [11]. *NPA* is very similar to *CD* and *SPA* is very similar to *DD*. *HPA* and *HPA-ELD* both have some similarities with *IDD*, as all 3 algorithms essentially eliminate the redundant computation inherent in *DD*. However, the approach taken in *HPA* (and *HPA-ELD*) is quite different than that taken in *IDD*. In pass k of *HPA*, for each transaction containing I items, $C = \binom{I}{k}$ potential candidates of size k are generated. Each of these potential candidates is hashed to determine which processor might contain the candidate itemset matching these potential candidate. These C potential candidates are sent only to the corresponding processors. Then each processor checks these potential candidates collected from all the processors against the locally stored subset of candidate itemsets. The distribution of the candidate itemsets over processors is determined by the hash function. This may make it difficult to ensure that each processor receives equal number of candidates. Furthermore, the number of potential candidates of size k generated for a transaction containing I items is $O(\binom{I}{k})$. Hence, for values of k greater than 2, *HPA* can have much larger communication volume than that for *DD* and *IDD*. For small values of k (e.g., $k = 2$), it is possible for *HPA* to incur smaller communication overhead than *IDD*.

Several researchers have proposed parallel formulations of association rule algorithms [12], [13], [14]. Park, Chen, and Yu proposed *PDM* [12], a parallel formulation of the serial association rule algorithm *DHP* [15]. *PDM* is similar in nature to the *CD* algorithm. In [14], Zaki et.al. presented a parallelization of a serial algorithm originally introduced in [16]. This serial algorithm is of entirely different nature than *Apriori*, hence its parallel formulations cannot be compared to the algorithms discussed in this paper.

IV. Performance Analysis

In this section, we analyze the amount of work done by each algorithm and the scalability of each algorithm. In this analysis, a parallel algorithm is considered scalable when the efficiency can be maintained as the number of processors is increased, provided that the problem size is also increased [9]. Let T_{serial} be the runtime of a serial algorithm and T_p be the runtime of a parallel algorithm. Efficiency [9] (E) of a parallel algorithm is

$$E = \frac{T_{serial}}{P \times T_p}$$

A parallel algorithm is scalable if $P \times T_p$ and T_{serial} remain of the same order [9]. The problem size (i.e., the serial runtime) for the *Apriori* algorithm increases either by increasing N or by increasing M (as a result of lowering the minimum support) in the algorithms discussed in Section III. Table III describes the symbols used in this section.

As discussed in Section II, each iteration of the algorithm consists of two steps: (i) candidate generation and hash tree construction (ii) computation of *subset* function for each transaction. The derivation of the runtime of the *subset* function is much more involved. Consider a transaction that has I items. During the k^{th} pass of the algorithm, this transaction has $C = \binom{I}{k}$ potential candidates that need to be checked against the candidate hash tree. Note that for a given transaction, if checking for one potential candidate leads to a visit to a leaf node, then all the candidates of this transaction are checked against the leaf node. As a result, if this node is revisited due to a different candidate from the same transaction, no checking needs to be performed. Clearly the total cost of checking at the leaf nodes is directly proportional to the number of distinct leaf nodes visited with the transaction. We assume that the average number of candidate item-sets at the leaf nodes is S. Hence the average number of leaf nodes in a hash tree is $L = M/S$. In the implementation of the algorithm, the desired value of S can be obtained by adjusting the branching factor of the hash tree. In general, the cost of traversal for each potential candidate will depend on the depth of the leaf node in the hash tree reached by the traversal. To simplify the analysis, we assume that the cost of each traversal is the same. Hence, the total traversal cost is directly proportional to C. For each potential candidate, we define $t_{travers}$ to be the cost associated with the traversal of the hash tree and t_{check} to be the cost associated with checking the candidate item-sets of the reached leaf node.

Note that the number of distinct leaves checked by a transaction is in general smaller than the number of potential candidates C. This is because different potential candidates may lead to the same leaf node. In general, if C is relatively large with respect to the number of leaf nodes

TABLE III
SYMBOLS USED IN THE ANALYSIS.

symbol	definition
N	Total number of transactions
P	Number of processors
M	Total number of candidates
G	Number of partitions of candidates in the *HD* algorithm
k	Pass number in Apriori algorithm
I	Average number of items in a transaction
C	Average number of potential candidates in a transaction
S	Average number of candidates at the leaf node
L	Average number of leaves in the hash tree for the serial *Apriori* algorithm
$t_{travers}$	Cost of hash tree traversal per potential candidate
t_{check}	Cost of checking at the leaf with S candidates
$V_{i,j}$	Expected number of leaves visited with i potential candidates and j leaves

in the hash tree, then the number of distinct leaf nodes visited will be smaller than C. We can compute the expected number of distinct leaf nodes visited as follows. To simplify the analysis, we assume that each traversal of the hash tree due to a different potential candidate is equally likely to lead to any leaf node of the hash tree.

Let P_v be the probability of reaching a previously visisted node and P_n be the probability of reaching a new node. Then, $V_{i,j}$, the expected number of distinct leaf nodes visited when the transaction has i potential candidates, and the hash tree has j leaf nodes is:

$$\begin{aligned} V_{1,j} &= 1 \\ V_{i,j} &= V_{i-1,j} \times P_v + (V_{i-1,j}+1) \times P_n \\ &= V_{i-1,j}\frac{V_{i-1,j}}{j} + (V_{i-1,j}+1)\frac{j-V_{i-1,j}}{j} \\ &= 1 + \frac{j-1}{j}V_{i-1,j} \\ &= \frac{1-\left(\frac{j-1}{j}\right)^i}{1-\left(\frac{j-1}{j}\right)} \\ &= \frac{j^i-(j-1)^i}{j^{i-1}} \qquad (1) \end{aligned}$$

Note that for large j, $V_{i,j} \simeq i$. This can be shown by taking limit on Equation 1:

$$\begin{aligned} \lim_{j\to\infty} V_{i,j} &= \lim_{j\to\infty} \frac{j^i-(j-1)^i}{j^{i-1}} \\ &= \frac{[i(i-1)\cdots 3\cdot 2]\, j - [i(i-1)\cdots 3\cdot 2]\,(j-1)}{(i-1)(i-2)\cdots 2\cdot 1} \\ &= ij - i(j-1) \\ &= i \qquad (2) \end{aligned}$$

This shows that if the hash tree size is much larger than the number of potential candidates in a transaction, then each potential candidate is likely to visit a distinct leaf node in the hash tree.

Serial *Apriori* algorithm. Recall that in the serial *Apriori* algorithm, the average number of leaf nodes in the hash tree is $L = M/S$. Hence the number of distinct leaf visited per transaction is $V_{C,L}$, and the computation time per transaction for visiting the hash tree is:

$$T_{trans} = C \times t_{travers} + V_{C,L} \times t_{check}$$

So the run time of the serial algorithm for processing N transactions is:

$$\begin{aligned} T^{serial}_{comp} &= \underbrace{N \times T_{trans}}_{\text{subset function}} + \underbrace{O(M)}_{\text{hash tree construction}} \\ &= N \times C \times t_{travers} + N \times V_{C,L} \times t_{check} \\ &\quad + O(M) \qquad (3) \end{aligned}$$

The *CD* algorithm. In the *CD* algorithm the entire set of candidates is replicated at each processor. Hence the average number of leaf nodes in the local hash tree at each processor is $L = M/S$, which is the same as in the serial *Apriori* algorithm. Thus the *CD* algorithm performs the same computation per transactions as the serial algorithm, but each processor handles only N/P number of transactions. Hence the run time of the *CD* algorithm is:

$$\begin{aligned} T^{CD}_{comp} &= \underbrace{\frac{N}{P} \times T_{trans}}_{\text{subset function}} + \\ &\quad \underbrace{O(M)}_{\text{hash tree construction}} + \underbrace{O(M)}_{\text{global reduction}} \\ &= \frac{N}{P} \times C \times t_{travers} + \frac{N}{P} \times V_{C,L} \times t_{check} + \\ &\quad O(M) \qquad (4) \end{aligned}$$

Comparing Equation 4 to Equation 3, we see that *CD* performs no redundant computation. In particular, both the time for traversal and for checking scales down by a factor of P.

However, the cost of hash tree construction is the same as the serial algorithm, and *CD* has additional cost of global reduction. Hence, $P \times T^{CD}_{comp}$ will grow as $O(PM)$ with respect to $O(M)$, whereas T^{serial}_{comp} grows only as $O(M)$. This shows that *CD* does not scale with respect to the increasing M. If M is too large to fit in the main memory, then the set of transaction needs to be read from the disk $\frac{M}{M_{capacity}}$ times, adding another $O(\frac{N}{P} \times \frac{M}{M_{capacity}})$ term to the runtime. On some architectures, this can be significant. But in our discussion in the rest of the paper, we will ignore this term.

The *DD* algorithm. In the *DD* algorithm, the number of candidates per processor is M/P, as the candidate set is partitioned. Hence the average number of leaf nodes in the local hash tree of each processor is L/P. Therefore, the number of distinct leaf nodes visited per transaction is $V_{C,\frac{L}{P}}$, and the computation time per transaction is:

$$T^{DD}_{trans} = C \times t_{travers} + V_{C,\frac{L}{P}} \times t_{check}$$

The number of transactions processed by each processor is N, as the transactions are shifted around the processors. Hence, the computation per processor of the *DD* algorithm is:

$$\begin{aligned} T^{DD}_{comp} &= N \times T^{DD}_{trans} + \underbrace{O(\frac{M}{P})}_{\text{hash tree construction}} + \underbrace{O(N)}_{\text{data movement}} \\ &= N \times C \times t_{travers} + N \times V_{C,\frac{L}{P}} \times t_{check} + \\ &\quad O(\frac{M}{P}) + O(N) \qquad (5) \end{aligned}$$

Comparing Equation 5 with the serial complexity (Equation 3), we see that the *DD* algorithm does not reduce the computation associated with the hash tree traversal. For both the serial *Apriori* and the *DD* algorithm, this cost is $N \times C \times t_{travers}$. However, the *DD* algorithm is able to reduce the cost associated with the checking at the leaf nodes. In particular, it reduces the serial cost of $N \times V_{C,L} \times t_{check}$ down to $N \times V_{C,\frac{L}{P}} \times t_{check}$. However, because $V_{C,\frac{L}{P}} > V_{C,L}/P$, the reduction achieved in this part is less than a factor of P. We can easily see this if we consider the case when L is very large. In this case, $V_{C,\frac{L}{P}} \simeq C$ and $V_{C,L}/P \simeq C/P$ by Equation 2. Thus, the number of leaf nodes checked over all the processors by the *DD* algorithm is higher than that of the serial algorithm. This is why the *DD* algorithm performs redundant computation.

Furthermore, *DD* has an extra cost of data movement. Due to these two factors, *DD* does not scale with respect to increasing N. However, the cost of building hash tree scales down by a factor of P. Thus, *DD* is scalable with respect to increasing M.

The *IDD* algorithm. In the *IDD* algorithm, just like the *DD* algorithm, the average number of leaf nodes in the local hash tree of each processor is L/P. However, the average number of potential candidates that need to be checked for each transaction at each processor is much less than *DD*, because of the intelligent partitioning of candidates set and the use of bitmap to prune at the root of the hash tree. More precisely, the number of potential candidates that need to be checked for a transaction is roughly C/P assuming that we have a good balanced partition. So the computation per transaction is:

$$T^{IDD}_{trans} = \frac{C}{P} \times t_{travers} + V_{\frac{C}{P},\frac{L}{P}} \times t_{check}$$

Thus the computation per processor is:

$$\begin{aligned} T^{IDD}_{comp} &= N \times T^{IDD}_{trans} + \underbrace{O(\frac{M}{P})}_{\text{hash tree construction}} + \underbrace{O(N)}_{\text{data movement}} \\ &= N \times \frac{C}{P} \times t_{travers} + N \times V_{\frac{C}{P},\frac{L}{P}} \times t_{check} + \\ &\quad O(\frac{M}{P}) + O(N) \qquad (6) \end{aligned}$$

Comparing Equation 6 to Equation 3, we see that the *IDD* algorithm is successful in reducing the cost associated with the hash tree traversal linearly. It also reduces the checking cost from $N \times V_{C,L} \times t_{check}$ down to $N \times V_{\frac{C}{P},\frac{L}{P}} \times t_{check}$. Note that for sufficiently large L, $V_{C,L} \simeq C$ and $V_{\frac{C}{P},\frac{L}{P}} \simeq C/P$. This shows that *IDD* is also able to linearly reduce the cost of checking at the leaf nodes, and thus unlike *DD*, it performs no redundant work. The comparison of *DD* and *IDD* in terms of the average number of distinct leaf node visited per transaction is reported in our experiment (see Figure 11 and discussions in Section V). However, P must be relatively small for *IDD* to have a good load balance. If P becomes large where M is fixed, the problem of load imbalance discussed in Section III makes some processors work on more than $1/P$ of items in a transaction at the root of the hash tree.

If the parallel architecture has hardware support for communication and computation to proceed concurrently and the amount of computation in the *subset* function is significant, the data movement cost in *IDD* can be made to be negligible. In the absence of such hardware support, the cost of data movement in *IDD* is $O(N)$. Thus IDD is not scalable with respect to N, but scales better than DD, as IDD does not have redundant computations. Like DD, IDD is also scalable with respect to increasing M.

The *HD* algorithm. In the *HD* algorithm, the number of potential candidates per transactions is C/G and the number of candidates per processor is M/G. So the computation time per transaction is:

$$T^{HD}_{trans} = \frac{C}{G} \times t_{travers} + V_{\frac{C}{G},\frac{L}{G}} \times t_{check}$$

The total number of transactions each processor has to process is GN/P. Thus the computation per processor is:

$$T^{HD}_{comp} = \frac{G \times N}{P} \times T^{HD}_{trans} + \underbrace{O(\frac{M}{G})}_{\text{hash tree construction}} +$$

$$\underbrace{O(\frac{G \times N}{P})}_{\text{data movement}} + \underbrace{O(\frac{M}{G})}_{\text{global reduction}}$$
$$= \frac{G \times N}{P} \times \frac{C}{G} \times t_{travers} + \frac{G \times N}{P} \times V_{\frac{C}{G},\frac{L}{G}} \times t_{check} + O(\frac{M}{G}) + O(\frac{G \times N}{P}) \quad (7)$$

Compared to the serial algorithm, Equation 7 shows that the *HD* algorithm reduces the computation linearly with respect to the hash tree traversal cost. The traversal cost is reduced from $N \times C \times t_{travers}$ down to $N \times C \times \frac{t_{travers}}{P}$. The cost of checking at the leaf nodes is reduced from $N \times V_{C,L} \times t_{check}$ down to $(G \times N \times V_{\frac{C}{G},\frac{L}{G}} \times t_{check})/P$. Note that for sufficiently large L, $N \times V_{C,L} \simeq NC$ and $G \times N \times V_{\frac{C}{G},\frac{L}{G}}/P \simeq N \times C/P$. Thus, the *HD* algorithm has a linear speedup with respect to the cost of checking at the leaf nodes.

HD also has data movement cost. However, when P is increased with increasing N, the cost is almost constant provided G is unchanged. Thus *HD* is scalable with respect to increasing N. Furthermore, *HD* scales with increasing M provided G is chosen such that $\frac{M}{G}$ is constant.

We make a comparison of *HD* and *CD* using Equations 4 and 7. Equation 4 can be roughly summarized as $O(\frac{N}{P}) + O(M)$, and Equation 7 can be similarly summarized as $O(G \times \frac{N}{P}) + O(\frac{M}{G})$. We show the condition where the run time for *HD* is less than that of *CD*, i.e.,

$$O(G \times \frac{N}{P}) + O(\frac{M}{G}) < O(\frac{N}{P}) + O(M)$$

Solving for G, which is the number of candidate partitions in *HDD*, gives the following:

$$1 \; < \; G \; < \; O(\frac{M \times P}{N}) \quad (8)$$

Equation 8 shows that when M is relatively larger than N, *HD* can outperform *CD* by selecting wide range of G values. This equation also shows that as N becomes relatively larger than M, *HD* can reduce G to have a performance advantage over *CD*. When N is very large compared to $M \times P$, *HD* can choose G to be 1 and becomes exactly same as *CD*.

V. Experimental Results

We implemented our parallel algorithms on a 128-processor Cray T3E and SP2 parallel computers. Each processor on the T3E is a 600 Mhz Dec Alpha (EV5), and has 512 Mbytes of memory. The processors are interconnected via a three dimensional torus network that has a peak unidirectional bandwidth of 430 Mbytes/second, and a small latency. For communication we used the message passing interface (MPI). Our experiments have shown that for 16 Kbyte messages we obtain a bandwidth of 303 Mbytes/second and an effective startup time of 16 microseconds. SP2 nodes consist of a Power2 processor clocked 66.7 MHz with 128 Kbytes data cache, 32 Kbytes instruction cache, 256-bit memory bus, 256 Mbytes real memory and 1 Gbytes virtual memory. The SP High Performance Switch (HPS) has a theoretical maximum bandwidth of 110 Mbytes/second.

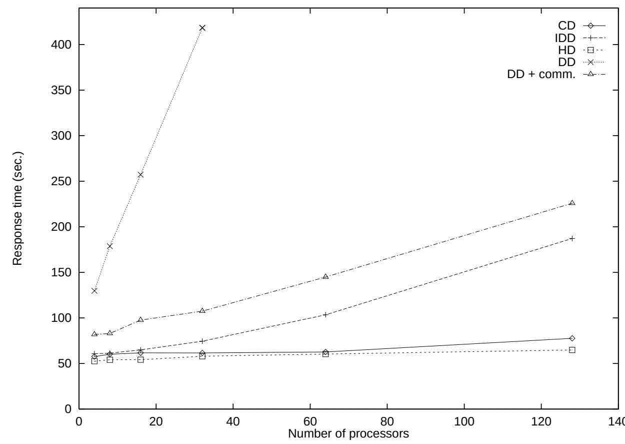

Fig. 10. Scaleup result on Cray T3E with 50K transactions and 0.1% minimum support.

We generated a synthetic dataset using a tool provided by [17] and described in [2]. The parameters for the data set chosen are average transaction length of 15 and average size of frequent item sets of 6. Data sets with 1000 transactions (63Kbytes) were generated for different processors. Due to the absence of a true parallel I/O system on the T3E system, we kept a set of transactions in a main memory buffer and read the transactions from the buffer instead of the actual disks. For the experiments involving larger data sets, we read the same data set multiple times. We also performed similar experiments on an IBM SP2 in which the entire database resided on disks. Our experiments (not reported here) show that the I/O requirements do not change the relative performance of the various schemes. We do present the results of one experiment on 16-processor SP2 for comparing *CD* to *IDD* and *HD* when *CD* scans database multiple times due to the partitioned hash tree.

To compare the scalability of the four schemes (*CD, DD, IDD* and *HD*), we performed scaleup tests with 50K transactions per processor and minimum support of 0.1%. With minimum support of 0.1%, the entire candidate hash tree fit in the main memory of one T3E processor. For this experiment, in the *HD* algorithm we have set the threshold on the number of candidates for switching to the *CD* algorithm to be 5K. With 0.1% support, the *HD* algorithm switched to *CD* algorithm in pass 5 of total 12 passes, and 88.4% of the overall response time of the serial code was spent in the first 4 passes. These scaleup results are shown in Figure 10.

As noted in [6], the *DD* algorithm scales very poorly. However, the performance achieved by *IDD* is much better than that of the *DD* algorithm. In particular, on 32 processors, *IDD* is faster than *DD* by a factor of 5.6. It can be seen that the performance gap between *IDD* and *DD* widens as the number of processors increases. *IDD* performs better

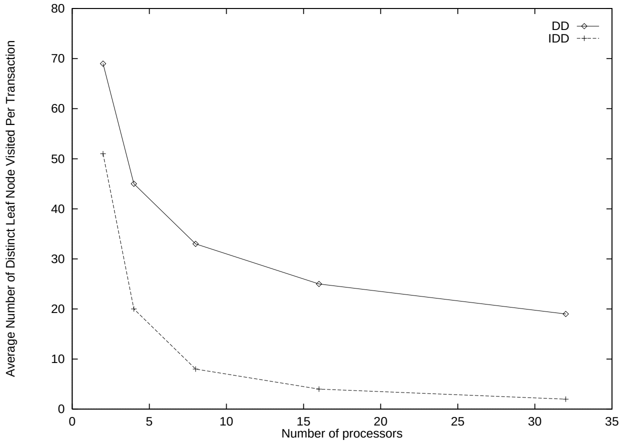

Fig. 11. Comparison of DD and IDD in terms of the average number of distinct leaf node visited per transaction with 50K transactions per processor and 0.2% minimum support.

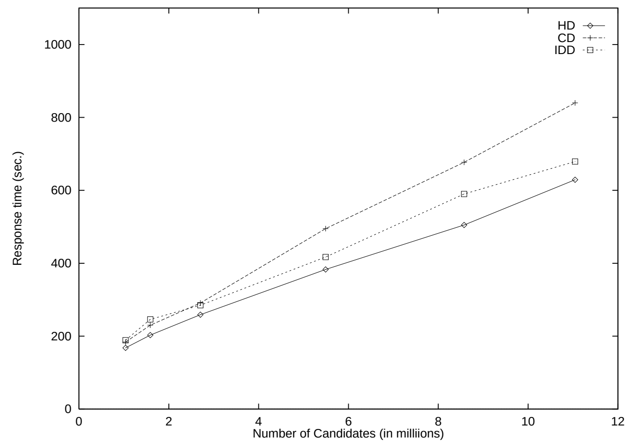

Fig. 12. Response time on 16 processor IBM SP2 with 100K transactions as the minimum support varies from 0.1% to 0.025%.

than DD because of the better communication mechanism for data movements and the intelligent partitioning of the candidate set. To show the effects of these two improvements, we replaced the communication mechanism of the DD algorithm with that of the IDD. The scaleup result of this improvement is shown as "DD+comm" in Figure 10. Hence the response time reduction from DD to DD+comm is due to the the better communication mechanism for data movements, and the reduction from DD+comm to IDD is due to the intelligent partitioning of the candidate set. Same experiments of comparing DD, DD+comm, and IDD on IBM SP2 also showed the similar pattern. We also show the effect of IDD's intelligent partitioning over DD by actually counting the number of distinct leaf node visited by both algorithms. We wan to verify that the average number of distinct leaf node visited by IDD is indeed much less than DD. Figure 11 shows that $V_{\frac{C}{P},\frac{L}{P}}$ of IDD goes down by factor of P, but $V_{C,\frac{L}{P}}$ of DD does not go down by factor of P.

Note that the response time of IDD increases as we increase the number of processors. This is due to the load balancing problem discussed in Section III, where the number of candidates per processor decreases as the number of processors increases. Looking at the performance of the HD algorithm, we see that the response time remains almost constant as we increase the number of processors while keeping the number of transactions per processor and the minimum support fixed. Comparing against CD, we see that HD actually performs better as the number of processors increases. Its performance on 128 processors is 16.5% better than CD. This performance advantage of HD over CD is due to the smaller cost of building candidate hash tree and global reduction in HD.

In the previous experiment, we chose the minimum support high enough such that the entire candidate hash tree fits in main memory. When the candidate hash tree does not fit in main memory, CD partitions it such that each partition fits in the main memory. Now the entire set of local transactions have to be read at each processor as many times as the number of partitions. This method increases the I/O cost. On the system in which I/O is scalable and fast (e.g., IBM SP2), this cost may be acceptable. We implemented the CD algorithm to partition the hash tree and read database multiple times in case the hash tree does not fit into main memory. Figure 12 shows the performance comparison of CD, IDD and HD on 16-processor IBM SP2 machine as the number of candidates increases by lowering minimum support. Unlike the earlier experiments on Cray T3E machine, the whole transactions were read in from the file. Figure 12 shows that as the number of candidates increases both IDD and HD outperform CD. This is due to the cost of building candidate hash tree, increased I/O time required for multiple scan of the database and increased communication time required for global reduction operation of multiple partitions of the candidate frequencies. Note that even on IBM SP2, the penalty due to these overheads is about 8% for 1 million candidates, 11% for 3 million candidates and 25% for 11 million candidates. For this particular experiments, the overhead of building the hash tree was the dominant cost. However, on systems with slower I/O, the I/O penalty can be substantial in addition to the overhead of building the hash tree.

In order to study the scalability of these algorithms, we performed experiments on T3E with varying number of processors (P), candidates (M), and transactions (N). For these experiments, we measured performance for computing size 3 frequent item sets only, as the computation for size 3 item sets took more than 55% of the total run time.

Figure 13 shows the speedup of three algorithms as P is increased from 4 to 64 with $N = 1.3$ million and $M = 0.7$ million. Note that the whole candidate hash tree fit in main memory and thus CD algorithm read in transactions only once. The figure clearly shows that the HD algorithm achieves better speedup than CD and IDD, and the difference in performance increases for larger number of processors. The reason for CD's poor speedup is the serial bottleneck of hash tree construction and global reduction operation. For 4 processors, the time taken for hash tree construction is only 3.1% of the total runtime and the

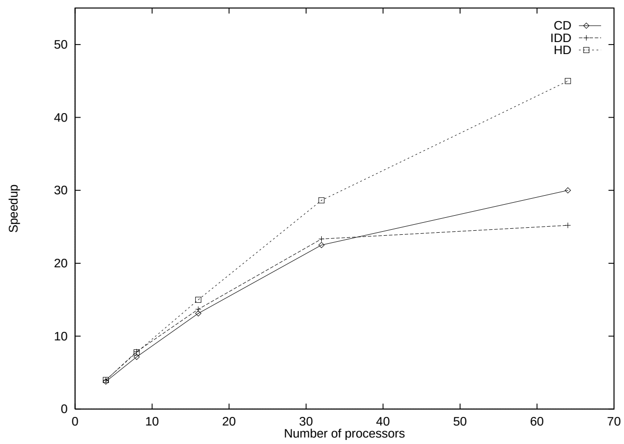

Fig. 13. Speedup of three algorithms on Cray T3E as P is increased from 4 to 64 with $N = 1.3$ million and $M = 0.7$ million. The processor configurations for *HD* were 8×2 for 16 processors, 8×4 for 32 processors, and 8×8 for 64 processors.

time for global reduction is only 1.6% of the total runtime. However, for 64 processors, these overheads are 24.8% and 31.0%, respectively. On the other hand, *IDD* has poor speedup due to the load imbalance, and data movement cost. For this particular experiment, the dominant overhead is load imbalance. In particular, for 4 processors the load imbalance overhead is only 6.3%, whereas for 64 processors this overhead is 49.6%. The cost of data movement is 1.0% for 4 processors and 6.4% for 64 processors. The processor configuration chosen for *HD* was 8×8 for 64 processors. Hence, *HD* performed one eighth of *CD*'s reduction operation and moved only one eighth of the data among groups of 8 processors only.

In the next experiment, we fixed P and M, and varied N from 1.3 million to 26.1 million. Figure 14 shows the runtime of this experiment. The figure shows that *CD* and *HD* scale nicely with the increasing number of transactions. However, with fixed M and P, *IDD* suffers from the load imbalance problem. In addition to that, the cost of data movement adds up as N is increased. However, this data movement cost is only 6.1% of the total runtime for 1.3 million candidate sets and 7.1% for 26.1 million candidate sets. Hence, the majority of runtime difference between *IDD* and the other two algorithms is due to the load imbalance.

The final experiment compares the runtime of three algorithms as M is increased from 0.7 million to 8.0 million with fixed N and P. The main memory of T3E was large enough to hold 0.7 million candidate sets. In *CD*, for the candidate size of greater than 0.7 million, the candidate set is partitioned and subset function was repeatedly called on the partitioned candidate sets. Figure 15 shows the runtime of this experiment. The figure shows that the performance gap between *CD* and *HD* widens as the number of candidate sets increases. This is due to the fact that *CD* has $O(M)$ component in its runtime. *HD* scales with respect to M as it has $O(\frac{M}{G})$ which is constant and $O(\frac{M}{P})$ as M becomes much larger. For smaller size of M, *IDD* performs worse than *CD*. As M increases, the performance

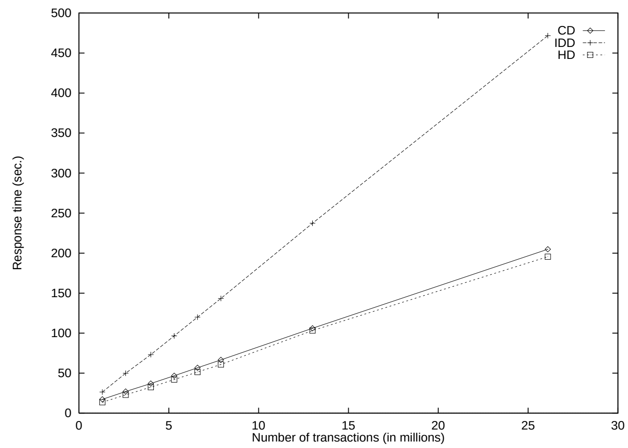

Fig. 14. Runtime of three algorithms on Cray T3E as N is increased from 1.3 million to 26.1 million with $M = 0.7$ million and $P = 64$. The processor configuration for *HD* was 8×8.

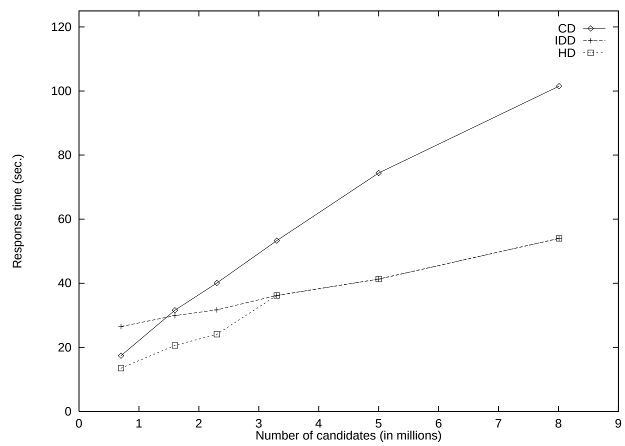

Fig. 15. Runtime of three algorithms on Cray T3E as M is increased from 0.7 million to 8.0 million with $N = 1.3$ million and $P = 64$. The processor configurations for *HD* were as follow. 8×8 for $M = 0.7$ million, 16×4 for $M = 1.7$ million, 32×2 for $M = 2.3$ million, and 64×1 for $M \geq 3.3$ million.

of *IDD* improves and eventually outperforms *CD*. This is due to the fact that *IDD* has $O(\frac{M}{P})$ component in its runtime compared to $O(M)$ of *CD*. Note that *HD* algorithm behaves exactly the same as *IDD* for the candidate set size of 3.3 million and more. This experiment shows that when M is much larger than N, *IDD* and *HD* are much better algorithms than *CD*.

For these experiments, just like the previous experiments on T3E, we simulated I/O and assumed that I/O cost is negligible compared to the computation cost. Even though *CD* algorithms repeatedly read transactions, no actual I/O was performed. However, when the I/O cost is factored in, the performance of *CD* would be worse than reported in these experiments.

VI. Conclusion

In this paper, we proposed two new parallel algorithms for mining association rules. The *IDD* algorithm effectively parallelizes the step of building hash tree, and is thus scalable with respect to the increasing candidate set size. This algorithm also utilizes total main memory available more effectively than the *CD* algorithm. This is important if the I/O cost becomes dominant due to slow I/O system. The *IDD* algorithm improves over the *DD* algorithm which has high communication overhead and redundant work. As shown in Section IV, for each transaction, the *DD* algorithm performs substantially more work overall than the serial *Apriori* algorithm. The communication and idling overheads were reduced using a better data movement communication mechanism, and redundant work was reduced by partitioning the candidate set intelligently and using bit maps to prune away unnecessary computation. Another useful feature of *IDD* is that it is well suited for the system environment with single source of data base. For instance, when all the data is coming from a database server or a single file system, one processor can read data from the single source and pass the data along the communication pipeline defined in the algorithm. However, as the number of available processors increases, the efficiency of this algorithm decreases due to load imbalance. Furthermore, *IDD* also suffers from $O(N)$ cost due to the communication of transactions, and hence is unscalable with respect to the number of transactions.

HD combines the advantages of *CD* and *IDD*. It is an improvement over CD, as it partitions the hash tree and thus avoids $O(M)$ cost of hash tree construction and global reduction. At the same time, it is an improvement over IDD, as it does not move data among all the processors, but only among a smaller subset of processors. Furthermore, *HD* achieves better load balancing than *IDD*, because the candidate set is partitioned into fewer buckets.

The experimental results on a 128-processor Cray T3E parallel machine show that the *HD* algorithm scales just as well as the *CD* algorithm with respect to the number of transactions, and scales as well as *IDD* with respect to increasing candidate set size. However, it outperforms *CD* when the number of candidate itemsets is large, and outperforms *IDD* when the number of transactions is very large.

Acknowledgments

This work was supported by NSF grant ASC-9634719, Army Research Office contract DA/DAAH04-95-1-0538, Cray Research Inc. Fellowship, and IBM partnership award, the content of which does not necessarily reflect the policy of the government, and no official endorsement should be inferred. Access to computing facilities was provided by AHPCRC, Minnesota Supercomputer Institute, Cray Research Inc., and NSF grant CDA-9414015.

References

[1] M. Stonebraker, R. Agrawal, U. Dayal, E. J. Neuhold, and A. Reuter, "DBMS research at a crossroads: The vienna update," in *Proc. of the 19th VLDB Conference*, Dublin, Ireland, 1993, pp. 688–692.

[2] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Proc. of the 20th VLDB Conference*, Santiago, Chile, 1994, pp. 487–499.

[3] M. A. W. Houtsma and A. N. Swami, "Set-oriented mining for association rules in relational databases," in *Proc. of the 11th Int'l Conf. on Data Eng.*, Taipei, Taiwan, 1995, pp. 25–33.

[4] A. Savasere, E. Omiecinski, and S. Navathe, "An efficient algorithm for mining association rules in large databases," in *Proc. of the 21st VLDB Conference*, Zurich, Switzerland, 1995, pp. 432–443.

[5] R. Srikant and R. Agrawal, "Mining generalized association rules," in *Proc. of the 21st VLDB Conference*, Zurich, Switzerland, 1995, pp. 407–419.

[6] R. Agrawal and J.C. Shafer, "Parallel mining of association rules," *IEEE Transactions on Knowledge and Data Eng.*, vol. 8, no. 6, pp. 962–969, December 1996.

[7] E.H. Han, G. Karypis, and V. Kumar, "Scalable parallel data mining for association rules," in *Proc. of 1997 ACM-SIGMOD Int. Conf. on Management of Data*, Tucson, Arizona, 1997.

[8] R. Agrawal, T. Imielinski, and A. Swami, "Mining association rules between sets of items in large databases," in *Proc. of 1993 ACM-SIGMOD Int. Conf. on Management of Data*, Washington, D.C., 1993.

[9] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis, *Introduction to Parallel Computing: Algorithm Design and Analysis*, Benjamin Cummings/ Addison Wesley, Redwod City, 1994.

[10] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[11] Takahiko Shintani and Masaru Kitsuregawa, "Hash based parallel algorithms for mining association rules," in *Proc. of the Conference on Paralellel and Distributed Information Systems*, 1996.

[12] J.S. Park, M.S. Chen, and P.S. Yu, "Efficient parallel data mining for association rules," in *Proceedings of the 4th Int'l Conf. on Information and Knowledge Management*, 1995.

[13] D. Cheung, V. Ng, A. Fu, and Y. Fu, "Efficient mining of association rules in distributed databases," *IEEE Transactions on Knowledge and Data Eng.*, vol. 8, no. 6, pp. 911–922, 1996.

[14] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li, "New parallel algorithms for fast discovery of association rules," *Data Mining and Knowledge Discovery: An International Journal*, vol. 1, no. 4, 1997.

[15] J.S. Park, M.S. Chen, and P.S. Yu, "An effective hash-based algorithm for mining association rules," in *Proc. of 1995 ACM-SIGMOD Int. Conf. on Management of Data*, 1995.

[16] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li, "New algorithms for fast discovery of association rules," in *Proc. of the Third Int'l Conference on Knowledge Discovery and Data Mining*, 1997.

[17] IBM Quest Data Mining Project, "Quest synthetic data generation code," *http://www.almaden.ibm.com/cs/quest/syndata.html*, 1996.