Scalable Parallel Data Mining for Association Rules *

Eui-Hong (Sam) Han
Department of Computer Science
University of Minnesota
Minneapolis, MN 55455
han@cs.umn.edu

George Karypis
Department of Computer Science
University of Minnesota
Minneapolis, MN 55455
karypis@cs.umn.edu

Vipin Kumar
Department of Computer Science
University of Minnesota
Minneapolis, MN 55455
kumar@cs.umn.edu

Abstract

One of the important problems in data mining is discovering association rules from databases of transactions where each transaction consists of a set of items. The most time consuming operation in this discovery process is the computation of the frequency of the occurrences of interesting subset of items (called candidates) in the database of transactions. To prune the exponentially large space of candidates, most existing algorithms, consider only those candidates that have a user defined minimum support. Even with the pruning, the task of finding all association rules requires a lot of computation power and time. Parallel computers offer a potential solution to the computation requirement of this task, provided efficient and scalable parallel algorithms can be designed. In this paper, we present two new parallel algorithms for mining association rules. The *Intelligent Data Distribution* algorithm efficiently uses aggregate memory of the parallel computer by employing intelligent candidate partitioning scheme and uses efficient communication mechanism to move data among the processors. The *Hybrid Distribution* algorithm further improves upon the *Intelligent Data Distribution* algorithm by dynamically partitioning the candidate set to maintain good load balance. The experimental results on a Cray T3D parallel computer show that the *Hybrid Distribution* algorithm scales linearly and exploits the aggregate memory better and can generate more association rules with a single scan of database per pass.

1 Introduction

One of the important problems in data mining [SAD+93] 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 extremely large amounts of data and large number of distinct items, the total number of candidates is prohibitively large. Hence, current association rule discovery techniques [AS94, HS95, SON95, SA95] try to prune the search space by requiring a minimum level of support for candidates under consideration. Support is a measure based on the number of occurrences of the candidates in database transactions. *Apriori* [AS94] is a recent state-of-the-art algorithm that aggressively prunes the set of potential candidates of size k by looking at the precise support for candidates of size $k-1$. 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.

Even with the highly effective pruning method of *Apriori*, the task of finding all association rules requires a lot of computation power that is available only in parallel computers. Furthermore, the size of the main memory in the serial computer puts an upper limit on the size of the candidate sets that can be considered in any iteration (and thus a lower bound on the minimum level of support imposed on candidates under consideration). Parallel computers also offer increased memory to solve this problem.

Two parallel algorithms, *Count Distribution* and *Data Distribution* were proposed in [AS96]. The *Count Distribution* algorithm has shown to scale linearly and have excellent speedup and sizeup behavior with respect to the number of transactions [AS96]. However, this algorithm works only when the entire hash tree in each pass of the algorithm fits into the main memory of single processor of the parallel computers. Hence, the *Count Distribution* algorithm, like its sequential counterpart *Apriori*, is unscalable with respect to increasing candidate size. The *Data Distribution* algorithm addresses the memory problem of the *Count Distribution* algorithm by partitioning the candidate set and assigning a partition to each processor in the system. However, this algorithm results in high communication overhead due to data movement and redundant computation [AS96].

In this paper, we present two parallel algorithms for mining association rules. We first present *Intelligent Data Distribution* algorithm that improves upon the *Data Distribu-*

*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. See http://www.cs.umn.edu/~han/papers.html#DataMiningPapers for an extended version of this paper and other related papers.

tion algorithm such that the communication overhead and redundant computation is minimized. The *Hybrid Distribution* algorithm further improves upon the *Intelligent Data Distribution* algorithm by dynamically grouping processors and partitioning the candidate set accordingly to maintain good load balance. The experimental results on a Cray T3D parallel computer show that the *Hybrid Distribution* algorithm scales linearly and exploits the aggregate memory better and can generate more association rules with a single scan of database per pass. An extended version of this paper that also contains the analysis of the performance of these schemes is available in [HKK97].

The rest of this paper is organized as follows. Section 2 provides an overview of the serial algorithm for mining association rules. Section 3 describes existing and proposed parallel algorithms. Experimental results are shown in Section 4. Section 5 contains conclusions.

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

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

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 $\{1\ 2\} \Longrightarrow \{3\}$, i.e. items 1 and 2 implies 3. The support of this rule is the frequency of the item-set $\{1\ 2\ 3\}$ in the transactions. For example, a support of 0.05 means that 5% of the transactions contain $\{1\ 2\ 3\}$. The confidence of this rule is defined as the ratio of the frequencies of $\{1\ 2\ 3\}$ and $\{1\ 2\}$. For example, if 10% of the transactions contain $\{1\ 2\}$, then the confidence of the rule is $0.05/0.10 = 0.5$. A rule that has a very high confidence (i.e., that is 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 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.

The task of discovering an association rule is to find all rules $X \overset{s,\alpha}{\Longrightarrow} Y$, where s is at least a given minimum support threshold and α is at least 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 has more support than the minimum support threshold specified) and the second step is to generate association rules that have higher confidence than the minimum confidence threshold from these frequent item-sets.

A number of algorithms have been developed for discovering association rules [AIS93, AS94, HS95]. Our parallel algorithms are based on the *Apriori* algorithm [AS94] 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 [AS94] for further details.

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

Figure 1: Apriori Algorithm

The *Apriori* algorithm consists of a number of passes. During pass k, the algorithm finds the set of frequent item-sets F_k of length k that satisfy the minimum support requirement. The algorithm terminates when F_k is empty. The high level structures of the *Apriori* algorithm are given in Figure 1. Initially F_1 contains all the items (i.e., item set of size one) that satisfy the minimum support requirement. Then for $k = 2, 3, 4, \ldots$, the algorithm generates C_k of candidates item-sets of length k using F_{k-1}. This is done in the function *apriori_gen*, which generates C_k by performing a join operation on the item-sets of 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 scan each transaction and see if it contains any of the candidate item-sets as its subset by performing a string-matching 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 [AS94]. Figure 2 shows one example of the candidate hash tree with candidates of length 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. When each candidate item-set is generated, the items in the set are stored in sorted order. Each candidate item-set is inserted into the hash tree by hashing each item at the internal nodes in sequence and following the links in the hash table. Once the 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 there are more items to be hashed in the candidate item-set, 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.

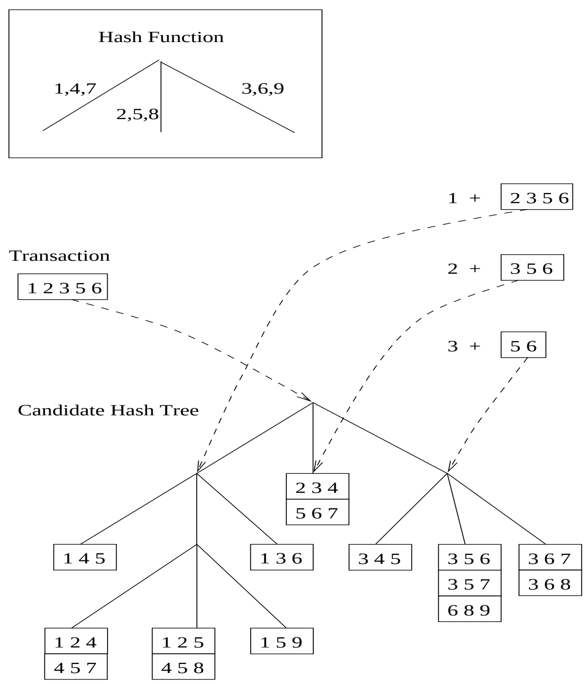

Figure 2: Subset operation on the root of a candidate hash tree.

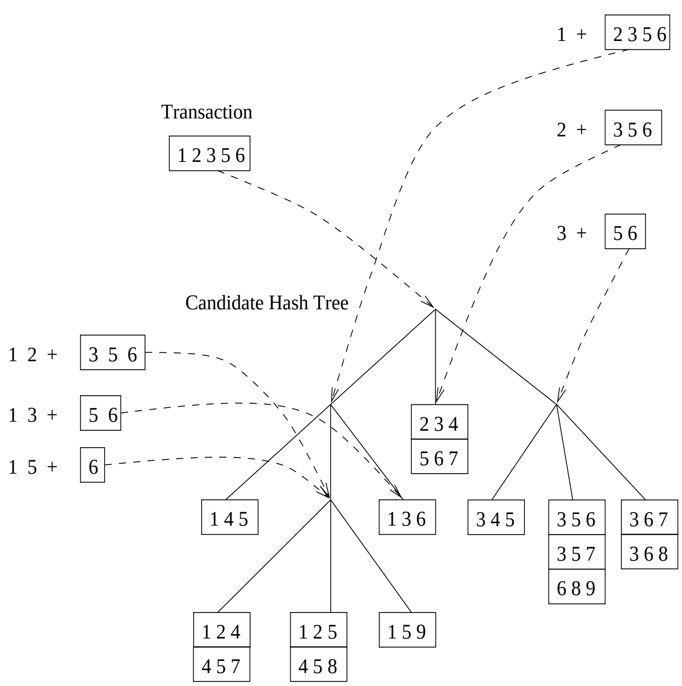

Figure 3: Subset operation on the left most subtree of the root of a candidate hash tree.

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.

The bulk of the computation is spent in finding the frequent item-sets and the amount of time required to find the rules from these frequent item-sets is relatively small. For this reason, parallel association algorithms focus on how to parallelize the first step. The parallel implementation of the second step is straightforward and is discussed in [AS96].

3 Parallel Algorithms

In this section, we will focus on the parallelization of the first task that finds all frequent item-sets. We first discuss two parallel algorithms proposed in [AS96] to help motivate our parallel formulations. In all our discussions, we assume that the transactions are evenly distributed among the processors.

3.1 Count Distribution Algorithm

In the *Count Distribution* (*CD*) algorithm proposed in [AS96], 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 [KGGK94]. 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 [AS96]. 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 works well only when the hash trees can fit into the main memory of each processor. 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. 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.

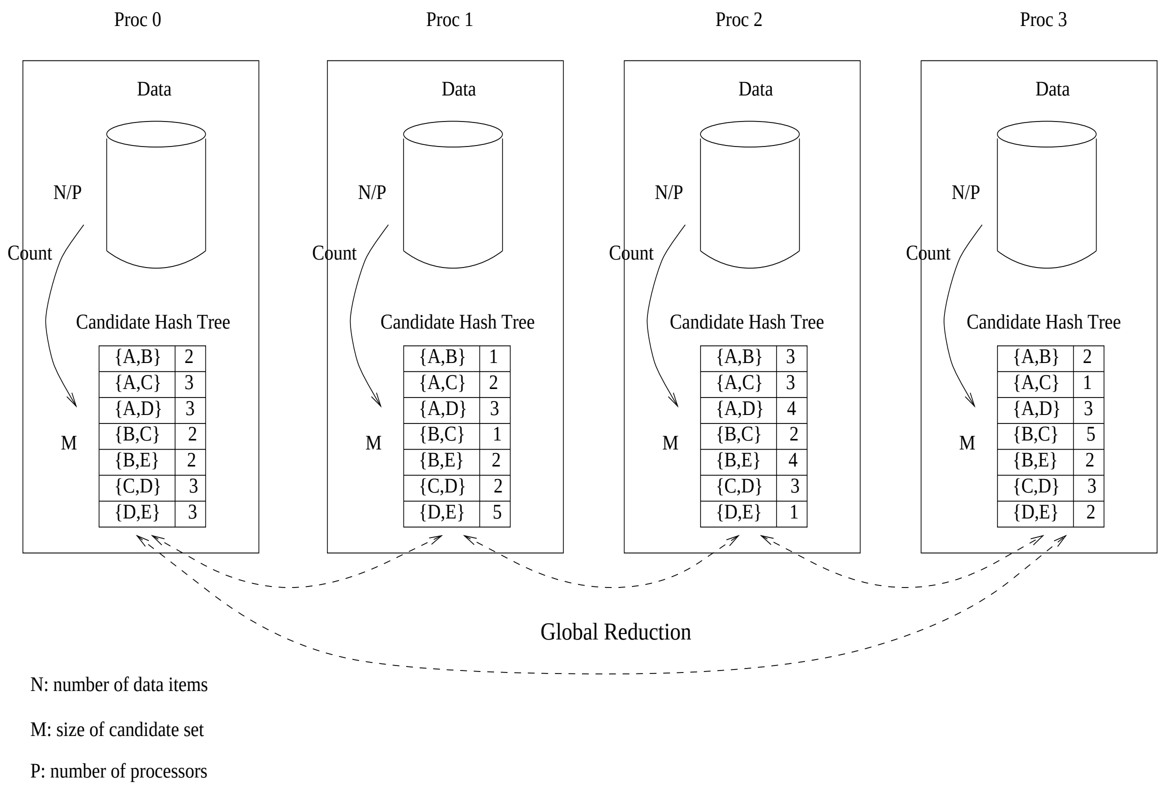

Figure 4: Count Distribution (CD) Algorithm

3.2 Data Distribution Algorithm

The *Data Distribution* (*DD*) algorithm [AS96] 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 according to the following fashion. 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, such that buffer j stores transactions from processor P_j. Now each processor P_i checks the P buffers to see which one contains data. Let k 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., $k = i$), then it is sent to all the other processors asynchronously 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., $k \neq i$), then it is cleared and an asynchronous receive request is issued to processor P_k. 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 [KGGK94]. 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 [AS96], 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 [AS96]. 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 two 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 $O(N)$. However, on

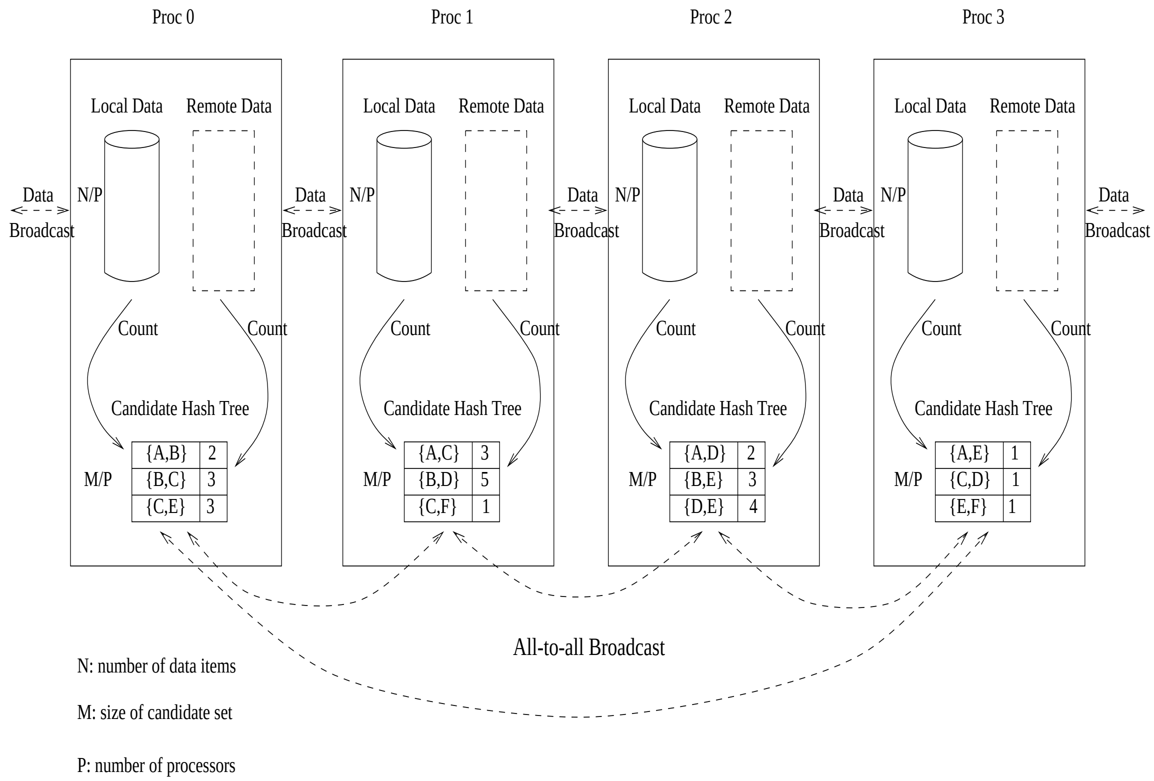

Figure 5: Data Distribution (DD) Algorithm

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.

Second, 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 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. 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. In *CD* (see Figure 4), only N/P transactions go through each hash tree of M candidates, whereas in *DD* (see Figure 5), N transactions have to go through each hash tree of M/P candidates. 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. However, for this to be true in the *DD* algorithm, the average depth of the hash tree has to be reduced by P and the average number of candidates in the leaf nodes has to be also reduced by P. This does not happen in the hash tree scheme discussed in Section 2. To see this, consider a hash tree with single candidate at the leaf node and with branching factor of B. By reducing the number of candidates by P, the depth of the hash tree will decrease by only $\log_B P$. With $B > P$ (which would be the most likely), the $\log_B P < 1$. On the other hand, when the hash tree is completely expanded to the depth k and the number of candidates at the leaf is greater than P, the number of candidates at the leaf goes down by P, but the depth of the tree does not change. In most of real cases, the hash tree will be in between these two extreme cases. However, in general, the amount of work per transaction will not go down by P of the original hash tree with M candidates.

3.3 Intelligent Data Distribution Algorithm

We developed the *Intelligent Data Distribution* (*IDD*) algorithm that solves the problems of the *DD* algorithm discussed in Section 3.2.

The locally stored portions of the database can be sent to all the other processors by using the ring-based all-to-all broadcast described in [KGGK94]. This operation does not suffer from the contention problems 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 proces-

```
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);
}
```

Figure 6: Pseudo Code for Data Movements

sor 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.

Recall that in the *DD* algorithm, the communication overhead of data movements dominates the computation work in the later passes of the process. In *IDD*, we solve this problem by switching to the *CD* algorithm when the total number of candidates falls below a threshold. Note that switching to the *CD* algorithm does not cause any communication or computation overhead since each processor can independently determine when to switch provided that the threshold parameter is globally known. A good choice of the parameter is the maximum number of candidates that single processor can have in the main memory.

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.

Figure 7 shows the high level picture of the algorithm. In this example, Processor 0 has all the candidates starting with items A and C, Processor 1 has all the candidates starting with B and E, 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 {A B C D E F G H} be a transaction that processor 0 is processing in the *subset* function discussed in Section 2. At the top level of the function, processor 0 will only proceed with items A and C (i.e., A + B C D E F G H and C + D E F G H). When the page containing this transaction is shifted to processor 1, this processor will only process items starting with B and E (i.e., B + C D E F G H and E + F G H). For each transaction in the database, our approach reduces the amount of work performed by each processor by a factor of P; thus, eliminates any redundant work. 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. 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. Another criteria is to have each processor have a mixed bag of the candidates. This will help to prevent load imbalance due to the skew in data. 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 a load-balanced distribution of the candidate item-sets, we use a partitioning algorithm that is based on bin-packing [PS82]. For each item in the database, we compute the number of candidate item-sets starting with this particular item. We then use a bin-packing algorithm to partition these items in P buckets such that the numbers of the candidate item-sets starting with these items in each bucket are equal. To remove any data skew, our bin-packing algorithm randomly selects the item to be assigned next in a bin. Figure 7 shows the partitioned candidate hash tree and its corresponding bitmaps in each processor. We were able to achieve less than 5% of load imbalance with the bin packing method described here.

3.4 Hybrid Algorithm

The *IDD* algorithm exploits the total system memory while minimizing the communication overhead involved. 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 data in SBuf of Figure 6. Eventually the amount of computation may be less than the communication involved, and this reduces overall efficiency. This will be an even more serious problem in a system that cannot

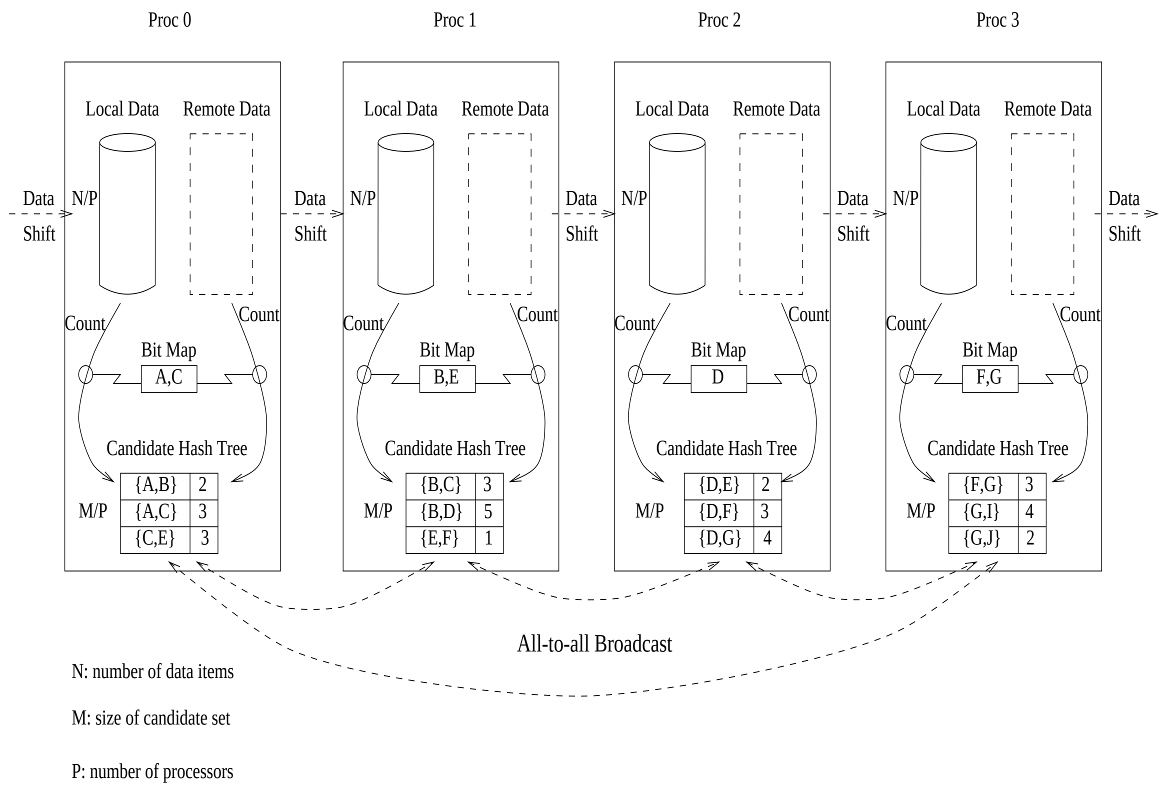

Figure 7: Intelligent Data Distribution (IDD) Algorithm

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, and the candidate set C_k is partitioned among the processors of each column of this grid. This partitioning of C_k is the same for each column of processors, that is, the processors along each row of the grid get the same subset of C_k. Now, the *IDD* algorithm is executed independently along each column of the grid, and the total counts of each subset of C_k is obtained by performing a reduction operation along the rows of this processor grid. Figure 8 illustrates the *HD* algorithm for a 3×4 grid of processors.

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. The same parameter that was used to determine whether to switch to *CD* algorithm can be used to decide the size of the partition in this algorithm. For example, let this parameter be C. If the total number of candidates M is less than C, it switches to *CD* algorithm. Otherwise find out the number of processor groups $G = \lceil M/C \rceil$ and form a logical $G \times P/G$ processor mesh configuration. In the example of Figure 8, 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 3.1. 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 corresponds to 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 in each of 4 hypothetical processors. This is shown in the step 1 of Figure 8. 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 counts of local candidates for all the data of the processors of the same column (or of a hypothetical processor). The global reduction operation is broken into two parts corresponding to the step 2 and 3 of the Figure 8. In the step 2, perform reduction operation [KGGK94] along the row such

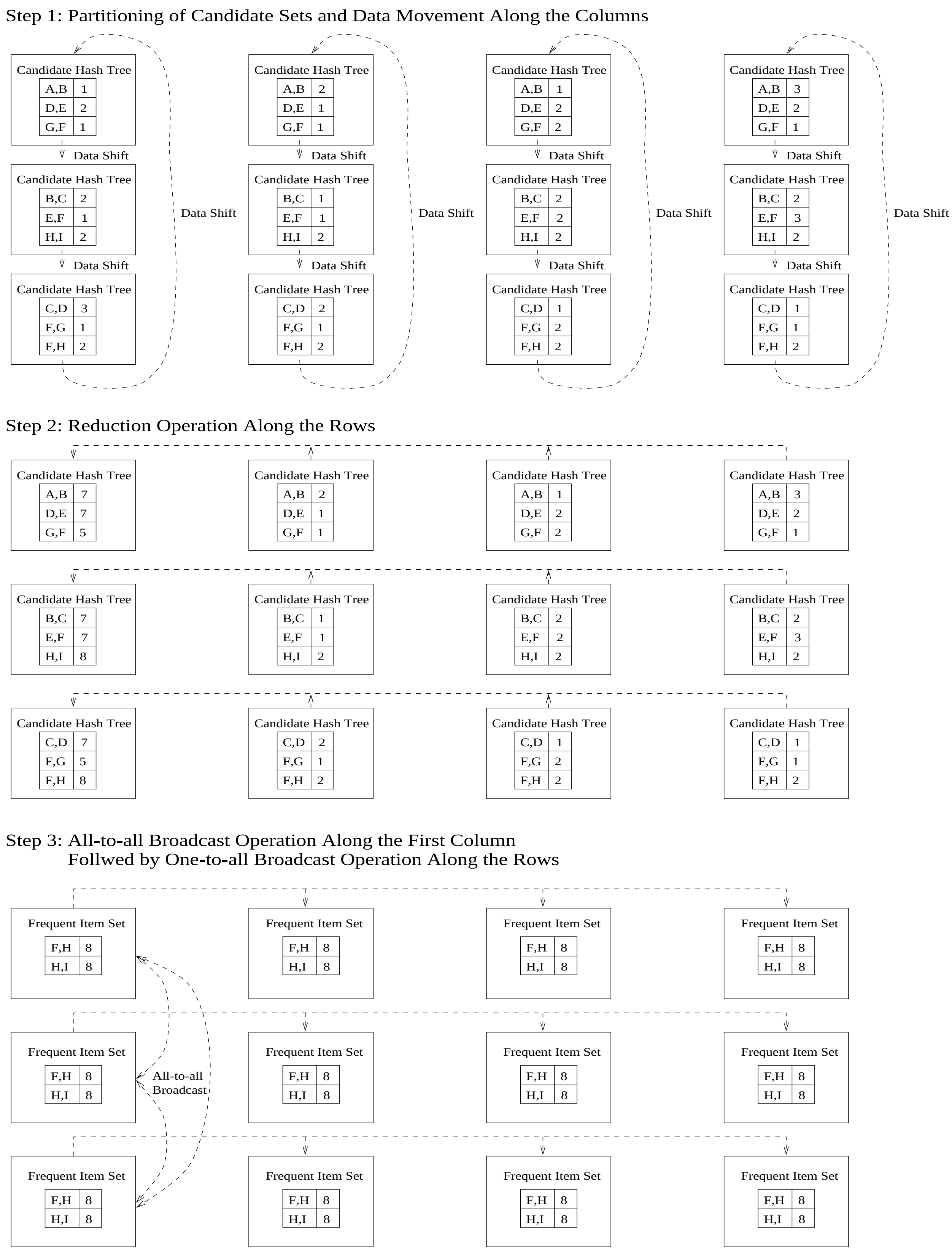

Figure 8: Hybrid Distribution (HD) Algorithm in 3×4 Processor Mesh ($G = 3, P = 12$)

that the processor in the first column of the same row has the total counts for the candidates in the same row processors. In the step 3, all the processors in the first column generate frequent set from the candidate set and perform all-to-all broadcast operation along the first column of the processor mesh. Then the processors in the first column broadcast the full frequent sets to the processors along the same row using one-to-all broadcast operation [KGGK94]. At this point, all the processors have the frequent sets and ready to proceed to the next pass.

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

4 Experimental Results

We implemented our parallel algorithms on a 128-processor Cray T3D parallel computer. Each processor on the T3D is a 150Mhz Dec Alpha (EV4), and has 64Mbytes of memory. The processors are interconnected via a three dimensional torus network that has a peak unidirectional bandwidth of 150Mbytes per second, and a small latency. For communication we used the message passing interface (MPI). Our experiments have shown that for 16Kbytes we obtain a bandwidth of 74Mbytes/seconds and an effective startup time of 150 microseconds.

We generated a synthetic dataset using a tool provided by [Pro96] and described in [AS94]. 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 (6.3KB) were generated for different processors. Due to the disk limitations of the T3D system we have kept the small transactions in the 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.[1]

We performed scaleup tests with 100K transactions per processor and minimum support of 0.25%. We could not use lower minimum support because the *CD* algorithm ran out of main memory. For this experiment, in the *IDD* and *HD* algorithms we have set the minimum number of candidates for switching to the *CD* algorithm very low to show the validity of our approaches. With 0.25% support, both algorithms switched to *CD* algorithm in pass 7 of total 12 passes and 90.7% of the overall response time of the serial code was spent in the first 6 passes. These scaleup results are shown in Figure 9.

As noted in [AS96], the *CD* algorithm scales very well. Looking at the performance obtained by *IDD*, we see that its response time increases as we increase the number of processors. This is due to the load balancing problem discussed in Section 3, where the number of candidates per processor decreases as the number of processors increases. However, the performance achieved by *IDD* is much better than that of the *DD* algorithm of [AS96]. In particular, *IDD* has 4.4 times less response time than *DD* on 32 processors. It can be seen that the performance gap between *IDD* and *DD* is widening as the number of processors increases. This is due to the improvement we made on *IDD* with the better communication mechanism for data movements and the intelligent partitioning of the candidate set. Looking at the performance of the *HD* algorithm, we see that 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 9.5% better than *CD*. This performance advantage of *HD* over *CD* is due to that the number of processors involved in global reduction operation of counts is much less in *HD* than in *CD*.

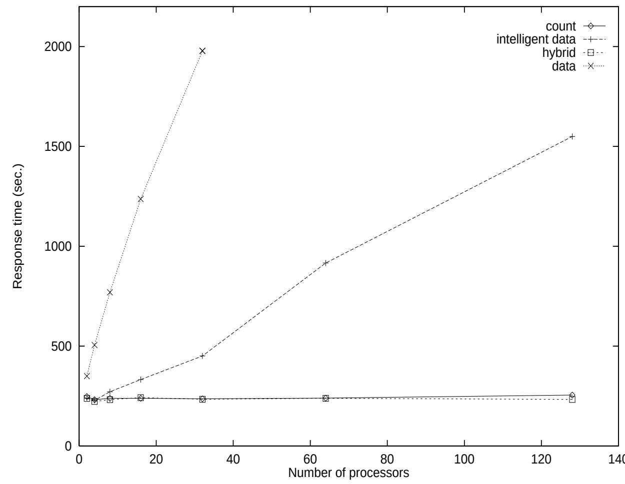

Figure 9: Scaleup result with 100K transactions and 0.25% minimum support.

We measured how our algorithms perform as we increase the number of transactions per processor from 50K(3.2MB) to 800K(50.4MB). For these experiments, we fixed the number of processors at 16 and the minimum support at 0.25%. These results are shown in Figure 10. From this figure, we can see that *CD* and *HD* perform almost identically. For both algorithms, the response time increases linearly with the number of transactions. *IDD* also scales linearly, but because of its load imbalance problem, its performance is somewhat worse.

Our experiments so far have shown that the performance of *HD* and *CD* are quite comparable. However, the real advantage of *HD* (and *IDD*) over *CD* is that they do not require the whole hash tree to reside on each processor, and thus better exploit the available memory. This allows us to use a smaller minimum support in the *Apriori* algorithm.

To verify this, we performed the experiments in which we fixed the number of transactions per processor to 50K and successively decreased the minimum support level. These experiments for 16 and 64 processors are shown in Figures 11 and 12 respectively. A couple of interesting observations can be made from these results. First, both *IDD* and *HD* successfully ran using lower support levels that *CD* could not run with. In particular, *IDD* and *HD* ran down to a support level of 0.06% on 16 processors and 0.04% on 64 processors. In contrast, *CD* could only run down to a support level of 0.25% and ran out of memory for the lower supports. The difference between the smaller support levels on 16 and 64 processors is due to the fact that the *IDD* and *HD* algorithms can exploit the aggregate memory of the larger number of processors.

[1] We also performed similar experiments on an IBM SP2 in which the entire database resided on disks. Our experiments show that the I/O requirements do not change the relative performance of the various schemes.

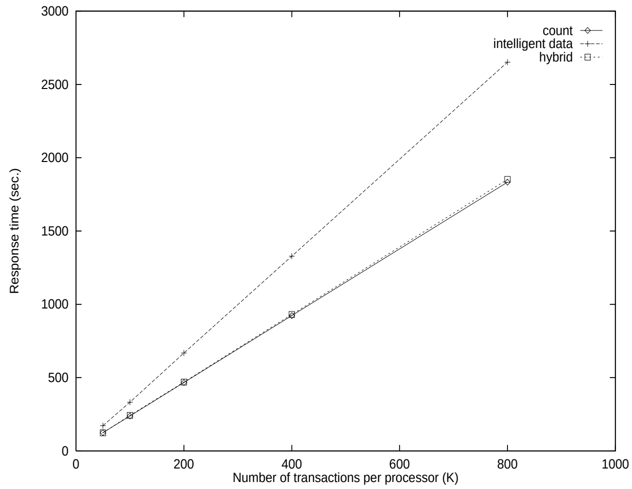

Figure 10: Sizeup result with 16 processors and 0.25% minimum support.

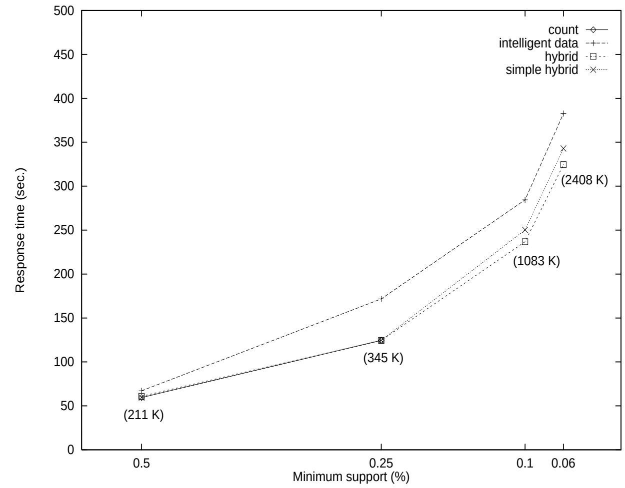

Figure 11: Response time on 16 processors with 50K transactions as the minimum support varies. At each support level, the total number of candidate item-sets is shown in parenthesis

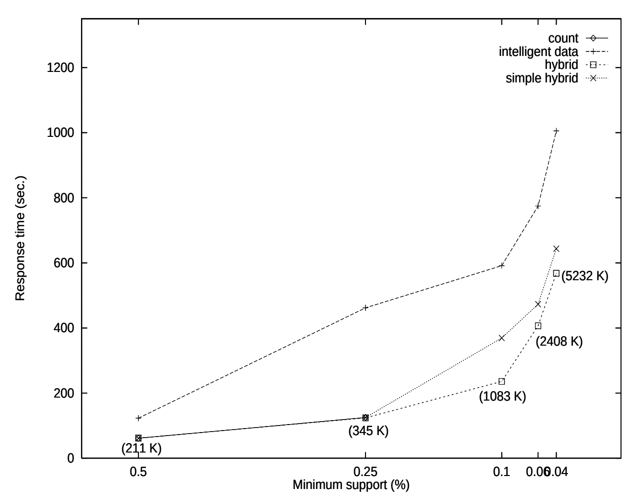

Figure 12: Response time on 64 processors with 50K transactions as the minimum support varies. At each support level, the total number of candidate item-sets is shown in parenthesis

The second thing to notice is that *HD* performs better than *IDD* both on 16 and 64 processors, and the relative performance of *IDD* compared to *HD* get worse as the number of processors increases. As discussed earlier, this performance difference is due to the load imbalance. As the number of processors increases, this load imbalance gets worse. However, on 16 processors *IDD* is 37% worse than *HD* for support level 0.25%, but only 18% worse for support of 0.06%. This is because as the support level decreases, the number of candidates (shown in parenthesis in Figures 11 and 12) increases which improves the load balance.

Figures 11 and 12 also show the performance of a simple hybrid algorithm obtained by combining *CD* and *IDD*. In this scheme, in each pass of the *Apriori* algorithm, we perform *CD* if the hash table can fit in the memory of each processors or *IDD* if it can not. As we can see from these results, this simple hybrid algorithm performs worse than *HD*. In particular, the relative performance of this scheme compare to *HD* gets worse as the number of processors increases. For example, for a support level of 0.06%, it is 6% worse on 16 processors and 17% worse on 64 processors. Thus the *HD* algorithm, by gradually adjusting the subsets of processors that perform *IDD* and *CD*, achieves better performance. This is because of the following two reasons. First, the candidate set is split among fewer number of processors which minimizes load imbalance and second, the reduction operation to obtain the counts in *CD* is performed among fewer processors, which decreases the communication overhead.

In another experiment, we varied the number of processors from 2 to 64 and measured how low we can go with minimum support for the *IDD* and *HD* algorithms. Table 1 shows the result for these algorithms. The result shows that as we have more processors, these algorithms can handle lower minimum support. Table 2 shows how the *HD* algorithm chose the processor configuration based on the number of candidates at each pass with 64 processors and 0.04% minimum support.

5 Conclusion

In this paper, we proposed two parallel algorithms for mining association rules. The *IDD* algorithm utilizes total main memory available more effectively than the *CD* algorithm.

Number of processors	1	2	4	8	16	32	64
Successful down to	0.25	0.2	0.15	0.1	0.06	0.04	0.03
Ran out of memory at	0.2	0.15	0.1	0.06	0.04	0.03	0.02

Table 1: Minimum support (%) reachable with different number of processors in our algorithms.

Pass	2	3	4	5	6	7	8	9	10
Configuration	8×8	64×1	4×16	2×32	2×32	2×32	2×32	2×32	1×64
No of Cand.	351K	4348K	115K	76K	56K	34K	16K	6K	2K

Table 2: Processor configuration and number of candidates of the *HD* algorithm with 64 processors and 0.04% minimum support for each pass. Note that 64×1 configuration is the same as the *DD* algorithm and 1×64 is the same as the *CD* algorithm. The total number of pass was 13 and all passes after 9 had 1×64 configuration.

This algorithms improves over the *DD* algorithm which has high communication overhead and redundant work. The communication overhead was 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. However, as the number of processors available increases, the efficiency of this algorithm decreases unless the amount of work is increased by having more number of candidates.

The *HD* combines advantages of the *CD* and *IDD*. This algorithm partitions candidate sets just like the *IDD* to exploit the aggregate main memory, but dynamically determines the number of partitions such that the partitioned candidate set fits into the main memory of each processor and each processor has enough number of candidates for computation. It also exploits the advantage of the *CD* by just exchanging counts information and moving around the minimum number of transactions among the smaller subset of processors.

The experimental results on a 128-processor Cray T3D parallel machine show that the *HD* algorithm scales just as well as the *CD* algorithm with respect to the number of transactions. It also exploits the aggregate main memory better and thus is able to find out more association rules with much smaller minimum support with a single scan of database per pass. The *IDD* algorithm also outperforms the *DD* algorithm, but is not as scalable as *HD* and *CD*.

Future works include applying these algorithms to real data like retail sales transaction, mail order history database and World Wide Web server logs [MJHS96] to confirm the experimental results in the real life domain. We plan to perform experiments on different platforms including Cray T3E, IBM SP2 and SGI SMP clusters. We also plan on implementing our ideas in generalized association rules [HF95, SA95], and sequential patterns [MTV95, SA96].

References

[AIS93] 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.

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th VLDB Conference*, pages 487–499, Santiago, Chile, 1994.

[AS96] R. Agrawal and J.C. Shafer. Parallel mining of association rules. *IEEE Transactions on Knowledge and Data Eng.*, 8(6):962–969, December 1996.

[HF95] J. Han and Y. Fu. Discovery of multiple–level association rules from large databases. In *Proc. of the 21st VLDB Conference*, Zurich, Switzerland, 1995.

[HKK97] E.H. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. Technical Report TR-97-??, Department of Computer Science, University of Minnesota, M inneapolis, 1997.

[HS95] 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.*, pages 25–33, Taipei, Taiwan, 1995.

[KGGK94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Algorithm Design and Analysis*. Benjamin Cummings/ Addison Wesley, Redwod City, 1994.

[MJHS96] B. Mobasher, N. Jain, E.H. Han, and J. Srivastava. Web mining: Pattern discovery from world wide web transactions. Technical Report TR-96-050, Department of Computer Science, University of Minnesota, M inneapolis, 1996.

[MTV95] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering frequent episodes in sequences. In *Proc. of the First Int'l Conference on Knowledge Discovery and Data Mining*, pages 210–215, Montreal, Quebec, 1995.

[Pro96] IBM Quest Data Mining Project. Quest synthetic data generation code. *http://www.almaden.ibm.com/cs/quest/syndata.html*, 1996.

[PS82] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[SA95] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proc. of the 21st VLDB Conference*, pages 407–419, Zurich, Switzerland, 1995.

[SA96] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. of the Fifth Int'l Conference on Extending Database Technology*, Avignon, France, 1996.

[SAD+93] 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*, pages 688–692, Dublin, Ireland, 1993.

[SON95] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. of the 21st VLDB Conference*, pages 432–443, Zurich, Switzerland, 1995.