

RJ 10004 (02/01/96)
Computer Science/Mathematics

IBM Research Report


Parallel Mining of Association Rules: Design, Implementation and Experience

Rakesh Agrawal
John C. Shafer

IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties).

 Research Division
Almaden · T.J. Watson · Tokyo · Zurich

Parallel Mining of Association Rules: Design, Implementation and Experience

Rakesh Agrawal

John C. Shafer*

IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

ABSTRACT: We consider the problem of mining association rules on a shared-nothing multiprocessor. We present three parallel algorithms that represent a spectrum of trade-offs between computation, communication, memory usage, synchronization, and the use of problem-specific information. We describe the implementation of these algorithms on IBM POWERparallel SP2, a shared-nothing machine. Performance measurements from this implementation show that the best algorithm, *Count Distribution*, scales linearly and has excellent speedup and sizeup behavior. The results from this study, besides being of interest in themselves, provide guidance for the design of parallel algorithms for other data mining tasks.

*Also Department of Computer Science, University of Wisconsin, Madison.

1. Introduction

With the availability of inexpensive storage and the progress in data capture technology, many organizations have created ultra-large databases of business and scientific data, and this trend is expected to grow. A complementary technology trend is the progress in networking, memory, and processor technologies that has opened up the possibility of accessing and manipulating these massive databases in a reasonable amount of time. Data mining (also called knowledge discovery in databases) is the efficient discovery of previously unknown patterns in large databases. The promise of data mining is that it will deliver technology that will enable development of a new breed of decision-support applications.

Recently, there has been considerable research in designing data mining algorithms (see, for example, [3] [4] [5] [8] [7] [11] [10] [14] [12] [15] [16] [17] [18]). However, the work so far has been concentrated on designing serial algorithms. Since the databases to be mined are often very large (measured in gigabytes and even terabytes), parallel algorithms are required.

We present in this paper three parallel algorithms for mining association rules [2], an important data mining problem. These algorithms have been designed to investigate and understand the performance implications of a spectrum of trade-offs between computation, communication, memory usage, synchronization, and the use of problem-specific information in parallel data mining. Specifically,

1. The focus of the *Count Distribution* algorithm is on minimizing communication. It does so even at the expense of carrying out redundant duplicate computations in parallel.
2. The *Data Distribution* algorithm attempts to utilize the aggregate main memory of the system more effectively. It is a communication-happy algorithm that requires nodes to broadcast their local data to all other nodes.
3. The *Candidate Distribution* algorithm exploits the semantics of the particular problem at hand to reduce synchronization between the processors and has load balancing built into it.

These algorithms have been implemented on an IBM POWERparallel System SP2 (henceforth referred to simply as SP2), a shared-nothing machine [13]. We present measurements from this implementation to evaluate the effectiveness of the design trade-offs.

These results, besides being of interest in themselves, have larger applicability. The performance evaluation can provide guidance to the designers of parallel algorithms for other data mining tasks (e.g. multi-level association rules [10] [19], sequential patterns [5]). The lessons learnt also carry over to other machines with shared-nothing architectures (e.g. GAMMA [6], Teradata [20]).

The organization of the rest of the paper is as follows. Section 2 gives a brief review of the problem of mining association rules [2] and the Apriori algorithm [4] on which the proposed parallel algorithms

are based. Section 3 gives the description of the parallel algorithms. Section 4 presents the results of the performance measurements of these algorithms. Section 5 contains conclusions.

2. Overview of the Serial Algorithm

2.1. Association Rules

Given a set of transactions, where each transaction is a set of items, an association rule is an expression $X \Rightarrow Y$, where X and Y are sets of items. The intuitive meaning of such a rule is that transactions in the database which contain the items in X tend to also contain the items in Y . An example of such a rule might be that 98% of customers that purchase tires and auto accessories also buy some automotive services; here 98% is called the *confidence* of the rule. The *support* of the rule $X \Rightarrow Y$ is the percentage of transactions that contain both X and Y . The problem of mining association rules is to find *all* rules that satisfy a user-specified minimum support and minimum confidence [2]. Applications include cross-marketing, attached mailing, catalog design, loss-leader analysis, add-on sales, store layout, and customer segmentation based on buying patterns.

Problem Decomposition. The problem of mining association rules can be decomposed into two subproblems [2]:

1. Find all sets of items (*itemsets*) whose support is greater than the user-specified minimum support. Itemsets with minimum support are called *frequent* itemsets.¹
2. Use the frequent itemsets to generate the desired rules. The general idea is that if, say, $ABCD$ and AB are frequent itemsets, then we can determine if the rule $AB \Rightarrow CD$ holds by computing the ratio $conf = \text{support}(ABCD)/\text{support}(AB)$. If $conf \geq$ minimum confidence, then the rule holds. (The rule will have minimum support because $ABCD$ is frequent.)

Much of the research has been focussed on the first subproblem as the database is accessed in this part of the computation and several algorithms have been proposed [2] [4] [12] [14] [17] [18]. We review in Section 2.2 the apriori algorithm [4] on which our parallel algorithms are based.

We chose to base our parallel implementation on the Apriori algorithm because of its superior performance over the earlier algorithms [2] [12], as shown in [4]. We preferred Apriori over AprioriHybrid, a somewhat faster algorithm in [4], because AprioriHybrid is harder to parallelize; the performance of AprioriHybrid is sensitive to heuristically determined parameters. Furthermore, by counting candidates of multiple sizes in one pass, the performance of Apriori can be made to approximate that of AprioriHybrid. The algorithm in [14] is quite similar to Apriori and our parallelization

¹In our earlier papers [2] [4], itemsets with minimum support were called *large* itemsets. However, some readers associated “large” with the number of items in the itemset, rather than its support. So we are switching the terminology to *frequent* itemsets.

k -itemset	An itemset having k items.
L_k	Set of frequent k -itemsets (those with minimum support). Each member of this set has two fields: i) itemset and ii) support count.
C_k	Set of candidate k -itemsets (potentially frequent itemsets). Each member of this set has two fields: i) itemset and ii) support count.
P^i	Processor with id i
D^i	The dataset local to the processor P^i
DR^i	The dataset local to the processor P^i after repartitioning
C_k^i	The candidate set maintained with the Processor P^i during the k th pass (there are k items in each candidate)

Figure 1: Notation

```

 $L_1 := \{\text{frequent 1-itemsets}\};$ 
 $k := 2;$  //  $k$  represents the pass number
while (  $L_{k-1} \neq \emptyset$  ) do
begin
   $C_k :=$  New candidates of size  $k$  generated from  $L_{k-1}$ ;
  forall transactions  $t \in \mathcal{D}$  do
    Increment the count of all candidates in  $C_k$  that are contained in  $t$ ;
   $L_k :=$  All candidates in  $C_k$  with minimum support;
   $k := k + 1;$ 
end
Answer  $:= \bigcup_k L_k;$ 

```

Figure 2: Apriori Algorithm

techniques directly apply to this algorithm as well. The algorithm in [18] does not perform as well as Apriori on large datasets with a large number of items. The algorithm in [17] attempts to improve the performance of Apriori by using a hash filter. However, as we will see in Section 4.3, this optimization actually slows down the Apriori algorithm.

2.2. Apriori Algorithm

Figure 2 gives an overview of the Apriori algorithm, using the notation given in Figure 1. The first pass of the algorithm simply counts item occurrences to determine the frequent 1-itemsets. A subsequent pass, say pass k , consists of two phases. First, the frequent itemsets L_{k-1} found in the $(k-1)$ th pass are used to generate the candidate itemsets C_k , using the apriori candidate generation procedure described below. Next, the database is scanned and the support of candidates in C_k is counted. For fast counting, we need to efficiently determine the candidates in C_k contained in a given transaction t . A hash-tree data structure [4] is used for this purpose.

Candidate Generation. Given L_{k-1} , the set of all frequent $(k-1)$ -itemsets, we want to generate a superset of the set of all frequent k -itemsets. The intuition behind the apriori candidate generation procedure is that if an itemset X has minimum support, so do all subsets of X . For simplicity, assume

the items in each itemset are in lexicographic order.

Candidate generation takes two steps. First, in the **join** step, join L_{k-1} with L_{k-1} :

```

insert into  $C_k$ 
select  $p.item_1, p.item_2, \dots, p.item_{k-1}, q.item_{k-1}$ 
from  $L_{k-1} p, L_{k-1} q$ 
where  $p.item_1 = q.item_1, \dots, p.item_{k-2} = q.item_{k-2}, p.item_{k-1} < q.item_{k-1}$ ;

```

Next, in the **prune** step, delete all itemsets $c \in C_k$ such that some $(k-1)$ -subset of c is not in L_{k-1} .

For example, let L_3 be $\{\{1\ 2\ 3\}, \{1\ 2\ 4\}, \{1\ 3\ 4\}, \{1\ 3\ 5\}, \{2\ 3\ 4\}\}$. After the join step, C_4 will be $\{\{1\ 2\ 3\ 4\}, \{1\ 3\ 4\ 5\}\}$. The prune step will delete the itemset $\{1\ 3\ 4\ 5\}$ because the itemset $\{1\ 4\ 5\}$ is not in L_3 . We will then be left with only $\{1\ 2\ 3\ 4\}$ in C_4 .

3. Parallel Algorithms

We first present three parallel algorithms for the first subproblem — the problem of finding all frequent itemsets. We then give a parallel algorithm for the second subproblem — the problem of generating rules from frequent itemsets. Refer to Figure 1 for a summary of notation used in the algorithm descriptions. We use superscripts to indicate processor id and subscripts to indicate the pass number (also the size of the itemset).

The algorithms assume a shared-nothing architecture, where each of N processors has a private memory and a private disk. The processors are connected by a communication network and can communicate only by passing messages. The communication primitives used by our algorithms are part of the MPI (Message Passing Interface) communication library supported on the SP2 and are candidates for a message-passing communication standard currently under discussion [9]. Data is evenly distributed on the disks attached to the processors, i.e. each processor’s disk has roughly an equal number of transactions. We do not require transactions to be placed on the disks in any special way.

3.1. Algorithm 1: Count Distribution

This algorithm uses a simple principle of allowing “redundant computations in parallel on otherwise idle processors to avoid communication”. The first pass is special. For all other passes $k > 1$, the algorithm works as follows:

1. Each processor P^i generates the complete C_k , using the complete frequent itemset L_{k-1} created at the end of pass $k-1$. Observe that since each processor has the identical L_{k-1} , they will be generating identical C_k .

2. Processor P^i makes a pass over its data partition D^i and develops local support counts for candidates in C_k .
3. Processor P^i exchanges local C_k counts with all other processors to develop global C_k counts. Processors are forced to synchronize in this step.
4. Each processor P^i now computes L_k from C_k .
5. Each processor P^i independently makes the decision to terminate or continue to the next pass. The decision will be identical as the processors all have identical L_k .

In the first pass, each processor P^i dynamically generates its local candidate itemset C_1^i depending on the items actually present in its local data partition D^i . Hence, the candidates counted by different processors may not be identical and care must be taken in exchanging the local counts to determine global C_1 .

Thus, in every pass, processors can scan the local data asynchronously in parallel. However, they must synchronize at the end of each pass to develop global counts.

Performance Considerations. Steps 1-2 and 4-5 are similar to that of the serial algorithm. The non-obvious step is how processors exchange local counts to arrive at global C_k counts. We give details of how we implement this step efficiently, separately for passes $k > 1$ and pass 1.

Pass $k > 1$: Recall that the candidates are kept in a hash-tree to allow efficient counting when making a pass over the data (Section 2.2). To exchange local counts, each processor P^i asynchronously extracts its local counts for C_k into a count array `LCntArr`. Note that since C_k is identical for all processors, if every processor traverses C_k in exactly the same order, corresponding elements of the count arrays will correspond to identical candidate itemsets. We thus do not have to communicate itemsets themselves but only their counts. We also save on computation because we can sum these local counts using simple vector summation rather than having to compare and match candidates.

Having created `LCntArr`, processors now do `ReduceScatter()` communication to perform a partitioned vector-sum of the count arrays. Figure 3 shows the `ReduceScatter()` operation pictorially. As the result of this operation, processor P^i receives in the `PartGCntArr` receive buffer the global counts of all the items in the i th `LCntArr` partition of all the processors. The number of items in each partition, `PartSize`, will be `sizeof(LCntArr)/N`.

```
ReduceScatter(SendBuf=LCntArr, ReceiveBuf=PartGCntArr,
              BlockLen=PartSize, ReductionFunction=add)
```

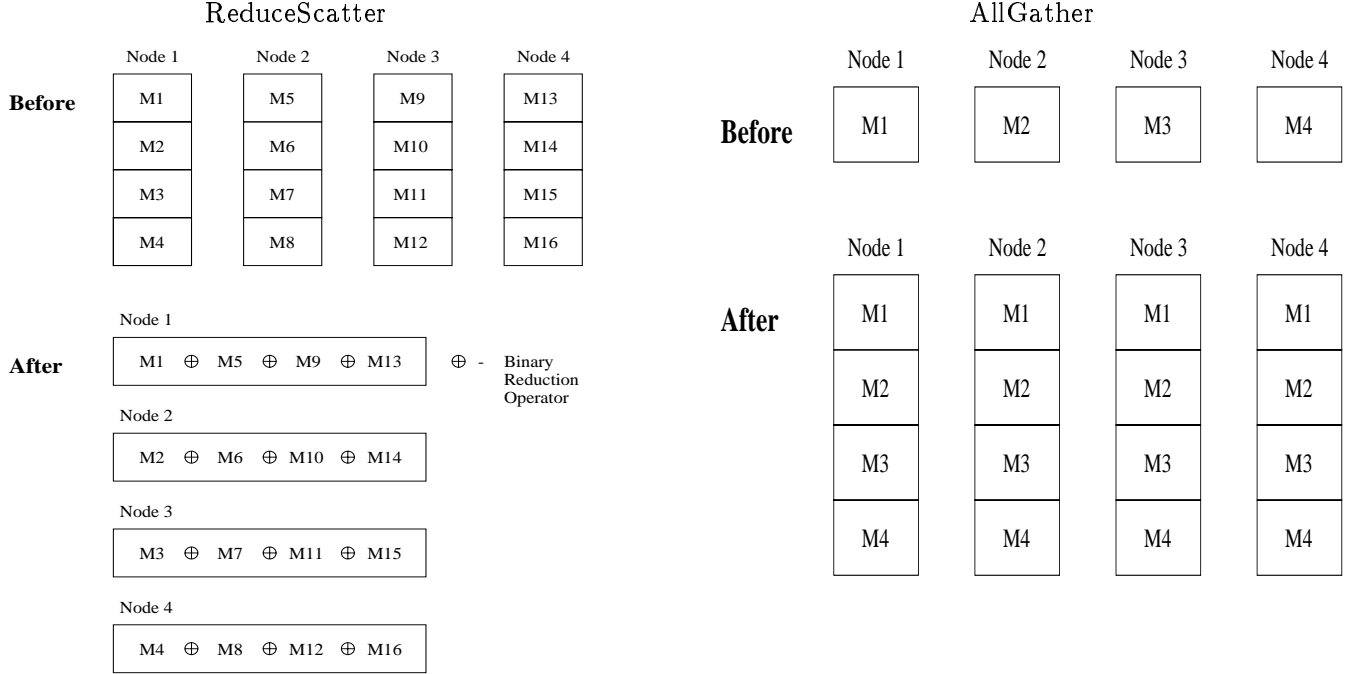


Figure 3: ReduceScatter and AllGather Communication

Each processor now gathers into GCntArr the global counts of items belonging to all other PartGCntArr partitions by calling

```
AllGather(SendBuf=PartGCntArr, ReceiveBuf=GCntArr, BlockLen=PartSize)
```

thus giving each processor the global counts for all candidates in C_k . Figure 3 shows the AllGather() operation pictorially.

Pass 1:. Each processor P^i makes a pass over its data partition D^i reading one tuple at a time and builds C_1^i , which is maintained in a closed hash-table. For each tuple, every item is hashed and its corresponding count in the hash table incremented; new entries are created if necessary.

At the end of the pass, processor P^i loads items and their counts from the hash table into a send buffer ItemsOfProcI and then gathers items and their support counts from all other processors. To do this, it must first gather the count of the total number of items residing in the send buffers of all other processors. Processor P^i puts the count of its own items in a CountBuf and calls

```
AllGather(SendBuf=CountBuf, ReceiveBuf=CountArr, BlockLen=sizeof(integer))
```

The j th element of the CountArr now contains the number of items processor j has in its send buffer.

Next, processor P^i calls `AllGatherV()`² to collect all items and their counts into the receive buffer `AllItems`³:

```
AllGatherV(SendBuf=ItemsOfProcI, ReceiveBuf=AllItems,
           BlockLen=sizeof(ItemsOfProcI), ReceiveBlockLen= CountArr)
```

P^i now hashes items from the receive buffer into a new hash table. If the same item was counted by more than one processor, it will hash to the same bucket and the support count for this item is accumulated. Thus, P^i now has the entire candidate set C_1 , complete with global counts.

3.2. Algorithm 2: Data Distribution

The attractive feature of the Count distribution algorithm is that no data tuples are exchanged between processors — only counts are exchanged. Thus, processors can operate independently and asynchronously while reading the data. However, the disadvantage is that this algorithm does not exploit the aggregate memory of the system effectively. Suppose that each processor has memory of size $|M|$. The number of candidates that can be counted in one pass is determined by $|M|$. As we increase the number of processors from 1 to N , the system has $N \times |M|$ total memory, but we still count the same number of candidates in one pass, as each processor is counting identical candidates.

The Data distribution algorithm is designed to exploit better the total system’s memory as the number of processors is increased. In this algorithm, each processor counts mutually exclusive candidates. Thus, as the number of processors is increased, a larger number of candidates can be counted in a pass. The downside of this algorithm is that every processor must broadcast its local data to all other processors in every pass. Therefore, this algorithm can become viable only on a machine with very fast communication.

Pass 1: Same as the Count distribution algorithm.

Pass $k > 1$:

1. Processor P^i generates C_k from L_{k-1} . It retains only $1/N$ th of the itemsets forming the candidates subset C_k^i that it will count. Which $1/N$ itemsets are retained is determined by the processor id and can be computed without communicating with other processors. In our implementation, itemsets are assigned in a round-robin fashion. The C_k^i sets are all disjoint and the union of all C_k^i sets is the original C_k .

²`AllGatherV()` is the variable length counterpart of `AllGather()` in which a processor receives messages of different sizes from other processors. `SendBuf` is of size `BlockLen`, `ReceiveBuf` is an array of N messages, and the size of the i th receive buffer is given by the i th element of the `ReceiveBlockLen` array.

³If `AllItems` array becomes too large, we have an intermediate step using `ReduceScatter()` to reduce the number of duplicate entries. We omit this detail for brevity.

2. Processor P^i develops support counts for the itemsets in its local candidate set C_k^i using both local data pages and data pages received from other processors.
3. At the end of the pass over the data, each processor P^i calculates L_k^i using the local C_k^i . Again, all L_k^i sets are disjoint and the union of all L_k^i sets is L_k .
4. Processors exchange L_k^i so that every processor has the complete L_k for generating C_{k+1} for the next pass. This step requires processors to synchronize. Having obtained the complete L_k , each processor can independently (but identically) decide whether to terminate or continue on to the next pass.

Performance Considerations. Steps 1 and 3 are straightforward. Step 4 for exchanging L_k^i is similar to the step 3 for exchanging C_1^i described with the first pass of the Count distribution algorithm, except that each L_k^i is disjoint and counts do not need to be summed. Each processor P^i loads L_k^i into a communication buffer, uses `AllGatherV()` to share L_k^i with everyone else and ends up with the complete L_k .

The interesting step is Step 2 in which processors develop support counts for local candidates C_k^i asynchronously. Processor P^i develops counts for candidates in C_k^i by alternating between data pages received from other processors in the receive buffers `RBufj` and tuples in its local data partition D^i . First, processor P^i uses $N - 1$

```
AsynchReceive(ReceiveBuf=RBufj, BufSize=PAGESIZE, Sender=DONTCARE)
```

to post $N - 1$ asynchronous receive buffers for receiving a page worth of data from any processor (indicated by specifying `DONTCARE` for the `Sender`).

P^i gives priority to processing a page in a receive buffer over a local tuple to avoid network congestion. Once P^i starts processing a receive buffer, it processes all the tuples in it. It then reposts the buffer using `AsynchReceive()` unless the receive buffer contains the end-of-transmission (EOT) flag. An EOT flag indicates that the sender has no more data to transmit.

If no data page is available in any of the receive buffers, P^i processes a local tuple from its partition D^i . After processing a local tuple, P^i adds it to a send buffer `SBuf`. After D^i has been completely processed, P^i adds an EOT flag to `SBuf`. If `SBuf` becomes full or an EOT flag has been added to it, P^i broadcasts `SBuf` to all other processors using $N - 1$ `AsynchSend()` calls (The MPI communication library does not support asynchronous multisend).

3.3. Algorithm 3: Candidate Distribution

Both Count and Data distribution algorithms require processors to synchronize at the end of a pass to exchange counts or frequent itemsets respectively. If the workload is not perfectly balanced,

this can cause all the processors to wait for whichever processor finishes last in every pass. The Candidate distribution algorithm attempts to do away with the dependence between processors so that they may proceed independently without synchronizing at the end of every pass.

In some pass l , where l is heuristically determined, this algorithm divides the frequent itemsets L_{l-1} between processors in such a way that a processor P^i can generate a unique C_m^i ($m \geq l$) independent of all other processors ($C_m^i \cap C_m^j = \emptyset$, $i \neq j$). At the same time, data is selectively replicated so that a processor can count candidates in C_m^i independent of all other processors. The choice of the redistribution pass is a tradeoff between decoupling processor dependence as soon as possible and waiting until the itemsets become more easily and equitably partitionable. The partitioning algorithm exploits the semantics of the Apriori candidate generation procedure described in Section 2.2.

After this candidate distribution, the only dependence that a processor has on other processors is for pruning the local candidate set during the prune step of candidate generation. However, a processor does not wait for the complete pruning information to arrive from all other processors. During the prune step of candidate generation, it prunes the candidate set as much as possible using whatever information has arrived, and opportunistically starts counting the candidates. The late arriving pruning information can be used in subsequent passes. The algorithm is described below.

Pass $k < l$: Use either Count or Data distribution algorithm.

Pass $k = l$:

1. Partition L_{k-1} among the N processors such that L_{k-1} sets are “well balanced”. We discuss below how this partitioning is done. Record with each frequent itemset in L_{k-1} which processor has been assigned this itemset. This partitioning is identically done in parallel by each processor.
2. Processor P^i generates C_k^i , logically using only the L_{k-1} partition assigned to it. Note that P^i still has access to the complete L_{k-1} , and hence can use standard pruning while generating C_k^i in this pass.
3. P^i develops global counts for candidates in C_k^i and the database is repartitioned into DR^i at the same time. The details of this step are given below.
4. After P^i has processed all its local data and any data received from all other processors, it posts $N - 1$ asynchronous receive buffers to receive L_k^j from all other processors. These L_k^j are needed for pruning C_{k+1}^i in the prune step of candidate generation.
5. Processor P^i computes L_k^i from C_k^i and asynchronously broadcasts it to the other $N - 1$ processors using $N - 1$ asynchronous sends.

Pass $k > l$:

1. Processor P^i collects all frequent itemsets that have been sent to it by other processors. They are used in the pruning step of the candidate generation, but not the join step. Itemsets received from processor j could be of length $k - 1$, smaller than $k - 1$ (slower processor), or greater than $k - 1$ (faster processor). P^i keeps track for each processor P^j the largest size of the frequent itemsets sent by it. Receive buffers for the frequent itemsets are reposted after processing.
2. P^i generates C_k^i using the local L_{k-1}^i . Now it can happen that P^i has not received L_{k-1}^j from all other processors, so P^i needs to be careful at the time of pruning. It needs to distinguish an itemset (a $k - 1$ long subset of a candidate itemset) which is *not* present in any of L_{k-1}^j from an itemset that *is* present in some L_{k-1}^j but this set has not yet been received by processor P^i . It does so by probing L_{l-1} (remember that repartitioning took place in pass l) using the $l - 1$ long prefix of the itemset in question, finding the processor responsible for it, and checking if L_{k-1}^j has been received from this processor.
3. P^i makes a pass over DR^i and counts C_k^i . It then computes L_k^i from C_k^i and asynchronously broadcasts L_k^i to every other processor using $N - 1$ asynchronous sends.

Performance Considerations. A performance sensitive step in the above algorithm is how to do data repartitioning efficiently (Step 3 of pass $k = l$). We give next the details of how the processors develop global counts of candidates assigned to them and at the same time do data repartitioning.

As in the Data distribution algorithm, each processor P^i uses $N - 1$ `AsynchReceive()` to post $N - 1$ asynchronous receive buffers for receiving a page worth of data from any processor. Processor P^i develops counts for candidates in C_k^i by alternating between data pages received from other processors in the receive buffers `RBufj` and tuples in its local data partition D^i . P^i gives priority to processing a page in a receive buffer over a local tuple to avoid network congestion. Once P^i starts processing a page in a receive buffer, it processes all the tuples in it and then reposts it using `AsynchReceive` unless the receive buffer contains the end-of-transmission (EOT) flag. However, unlike Data distribution, P^i allocates $N - 1$ send buffers for sending tuples to specific processors as discussed below.

If no data page is available in any of the receive buffers, P^i processes a local tuple in its partition D^i . It first probes L_{k-1} to see which processors can use this tuple by finding the processors assigned to the frequent itemsets of size $k - 1$ contained in this tuple. It writes the tuple into the send buffers of the corresponding processors. If P^i itself is one of the processors interested in this tuple, it uses the tuple to increment counts in C_k^i . After D^i has been completely processed, P^i adds an EOT flag to all the send buffers. If a send buffer becomes full or an EOT flag has been added to it, P^i asynchronously sends this buffer to the corresponding processor.

When processing a tuple from a receive buffer, P^i increments the counts in C_k^i . It then writes this tuple to a local file DR^i , discarding any item in the tuple that did not contribute to the counting. Local tuples used to count C_k^i are also written in this manner to DR^i . P^i uses this repartitioned DR^i in future passes to count the support for candidates.

Partitioning L_k . We motivate the algorithm for partitioning L_k by an example. Let L_3 be $\{ABC, ABD, ABE, ACD, ACE, BCD, BCE, BDE, CDE\}$. Then $L_4 = \{ABCD, ABCE, ABDE, ACDE, BCDE\}$, $L_5 = \{ABCDE\}$, and $L_6 = \emptyset$. Consider $\mathcal{E} = \{ABC, ABD, ABE\}$ whose members all have the common prefix AB . Note that the candidates $ABCD, ABCE, ABDE$ and $ABCDE$ also have the prefix AB . The apriori candidate generation procedure (Section 2.2) generates these candidates by joining only the items in \mathcal{E} .

Therefore, assuming that the items in the itemsets are lexicographically ordered, we can partition the itemsets in L_k based on common $k - 1$ long prefixes. By ensuring that no partition is assigned to more than one processor, we have ensured that each processor can generate candidates independently (ignoring the prune step). Suppose we also repartition the database in such a way that any tuple that supports an itemset contained in any of the L_k partitions assigned to a processor is copied to the local disk of that processor. The processors can then proceed completely asynchronously.

The actual algorithm is more involved because of two reasons. A processor may have to obtain frequent itemsets computed by other processors for the prune step of the candidate generation. In the example above, the processor assigned the set \mathcal{E} has to know whether $BCDE$ is frequent to be able to decide whether to prune the candidate $ABCDE$, but the set with prefix BC may have been assigned to a different processor. The other problem is that we need to balance load across processors. Details of the full partitioning algorithm are given in Appendix A⁴.

3.4. Parallel Rule Generation

We now present our parallel implementation of the second subproblem – the problem of generating rules from frequent itemsets. Generating rules is much less expensive than discovering frequent itemsets as it does not require examination of the data.

Given a frequent itemset l , rule generation examines each non-empty subset a and generates the rule $a \Rightarrow (l - a)$ with support = $support(l)$ and confidence = $support(l)/support(a)$. This computation can efficiently be done by examining the largest subsets of l first and only proceeding to smaller subsets if the generated rules have the required minimum confidence [4]. For example, given a frequent itemset $ABCD$, if the rule $ABC \Rightarrow D$ does not have minimum confidence, neither will $AB \Rightarrow CD$, and so we need not consider it.

⁴The paper is self-contained without this appendix. Should space become a constraint, the conference version of the paper will not include the appendix, but will refer to an IBM Research Report.

Generating rules in parallel simply involves partitioning the set of all frequent itemsets among the processors. Each processor then generates rules for its partition only using the algorithm above. Since the number of rules that can be generated from an itemset is sensitive to the itemset’s size, we attempt equitable balancing by partitioning the itemsets of each length equally across the processors.

Note that in the calculation of the confidence of a rule, a processor may need to examine the support of an itemset for which it is not responsible. For this reason, each processor must have access to all the frequent itemsets before rule generation can begin. This is not a problem for the Count and Data distribution algorithms because at the end of the last pass, all the processors have all the frequent itemsets. In the Candidate distribution algorithm, fast processors may need to wait until slower processors have discovered and transmitted all of their frequent itemsets. For this reason and because the rule generation step is relatively cheap, it may be better in the Candidate distribution algorithm to simply discover the frequent itemsets and generate the rules off-line, possibly on a serial processor. This would allow processors to be freed to run other jobs as soon as they are done finding frequent itemsets, even while other processors in the system are still working.

4. Performance Evaluation

We ran all of our experiments on a 32-node IBM SP2 Model 302. Each node in the multiprocessor is a Thin Node 2 consisting of a POWER2 processor running at 66.7MHz with 256MB of real memory. Attached to each node is a 2GB disk of which less than 500MB was available for our tests. The processors all run AIX level 3.2.5 and communicate with each other through the High-Performance Switch with HPS-2 adaptors. The combined communication hardware has a rated peak bandwidth of 80 megabytes per second and a latency of less than 40 microseconds. In our own tests of the base communication routines, actual point-to-point bandwidth reached 20MB/s. Experiments were run on an otherwise idle system. See [13] for further details of the SP2 architecture.

Name	T	I	\mathcal{D}_1	\mathcal{D}_{16}	\mathcal{D}_{32}	
D3278K.T5.I2	5	2	3278K	52448K	104896K	
D2016K.T10.I2	10	2	2016K	32256K	64512K	T Average transaction length
D2016K.T10.I4	10	4	2016K	32256K	64512K	I Average size of frequent itemsets
D1456K.T15.I4	15	4	1456K	23296K	46592K	\mathcal{D} Average number of transactions
D1140K.T20.I4	20	4	1140K	18240K	36480K	
D1140K.T20.I6	20	6	1140K	18240K	36480K	

Table 1: Data Parameters

We used synthetic datasets of varying complexity, generated using the procedure described in [4]. The characteristics of the six datasets we used are shown in Table 1. These datasets vary from many short transactions with few frequent itemsets, to fewer larger transactions with many frequent itemsets. All the datasets were about 100MB per processor in size. We could not use larger datasets

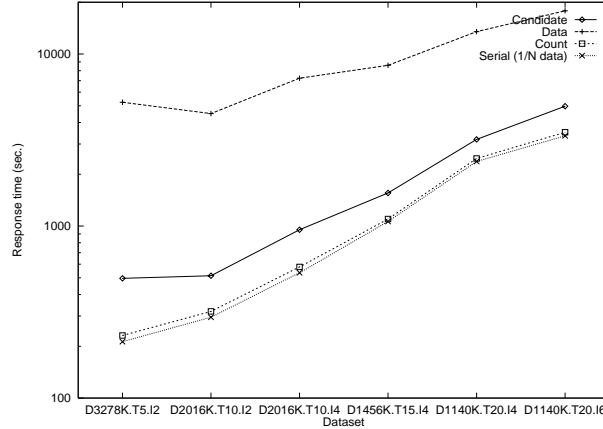


Figure 4: Relative Performance of the Algorithms

due to constraints on the amount of storage available on local disks; the Candidate algorithm writes the redistributed database on local disks after candidate partitioning, and we run out of disk space with the larger datasets. However, we include results of sizeup experiments (up to 400 MB per processor) for the Count distribution algorithm to show the trends for larger amounts of data per processor. Experiments were repeated multiple times to obtain stable values for each data point.

4.1. Relative Performance and Trade-offs

Figure 4 shows the response times for the three parallel algorithms on the six datasets on a 16 node configuration with a total database size of approximately 1.6GB. The response time was measured as the time elapsed from the initiation of the execution to the end time of the last processor finishing the computation. The response times for the serial version are for the run against only one node’s worth of data or 1/16th of the total database. We did not run the serial algorithm against the entire data because we did not have enough disk space available. We obtained similar results for other node configurations and dataset sizes. In the experiments with Candidate distribution, repartitioning was done during the fourth pass. In our tests, this choice yielded the best performance.

The results are very encouraging; for both Count and Candidate distribution algorithms, response times are close to that of the serial algorithm; this is especially true for Count. The overhead for Count is less than 7.5% when compared to the serial version run with 1/N data. Of that 7.5% overhead, about 2.5% was spent waiting for other processors to synchronize.

Among the parallel algorithms, Data distribution did not fare as well as the other two. As we had expected, Data was indeed able to better exploit the aggregate memory of the multiprocessor and make fewer passes in the case of datasets with large average transaction and frequent itemset lengths (see Table 2). However, its performance turned out to be markedly lower for two reasons: extra communication and the fact that every node in the system must process every single database

Name	Serial	Count	Data	Candidate
D3278K.T5.I2	7	7	7	7
D2016K.T10.I2	7	7	7	7
D2016K.T10.I4	11	11	11	11
D1456K.T15.I4	13	13	11	13
D1140K.T20.I4	21	21	11	21
D1140K.T20.I6	23	23	14	23

Table 2: Number of Data Passes Required

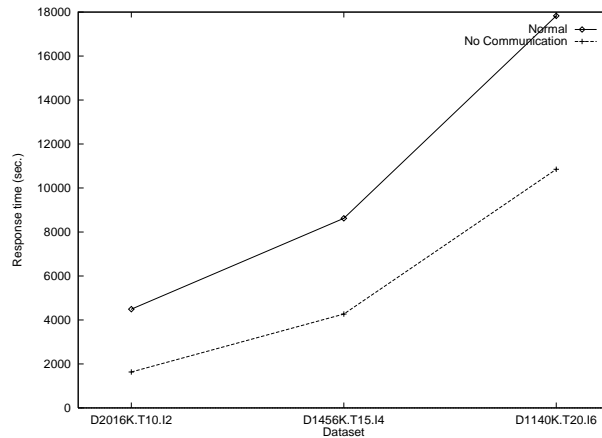


Figure 5: Communication Costs for Data Distribution

transaction. Communication is the worst of these two problems as show by Figure 5, even on a machine such as SP2 with very fast communication. The points labeled “Normal” correspond to the response times for the normal Data distribution algorithm on a 16-node configuration, but with the same 100MB of data replicated on each node. The points labeled “No Communication” correspond to a modified version of the Data distribution algorithm where, instead of receiving data from other nodes, a node simply processed its local data 15 more times. Since each node had the exact same data, this yielded the exact same results with the only difference being no time was spent on communication or its management. We did this for three of the six datasets and discovered that fully half of the time taken by Data distribution was for communication.

We had hoped for better results from the Candidate distribution algorithm, considering that it is the one that exploits the problem-specific semantics. Since the Candidate algorithm must also communicate the entire dataset during the redistribution pass, it suffers from the same problems as Data. Candidate, however, only performs this redistribution once. Also, unlike Data, processors may selectively filter out transactions it sends to other processors depending upon how the dependency graph is partitioned. This can greatly reduce the amount of data traveling through the network. Unfortunately, even a single pass of filtered data redistribution is costly. The question is whether or not the subsequent passes where each processor can run independently without synchronizing can compensate for this cost. As the performance results show, redistribution simply costs too much

relative to the cost of synchronization, making Count the winner.

Synchronization costs can become quite large if the data distributions are skewed or the nodes are not equally capable (different memory sizes, processor speeds, I/O bandwidths and capacities). Investigation of these issues is a broad topic and it is in our future plans. However, one can think of several alternatives for adding load balancing to the Count distribution algorithm that do not require redistribution of the complete database as in the case of the Candidate distribution algorithm. Extrapolating from the results of this study, our sense is that the Count distribution algorithm embellished with an appropriate load balancing strategy is likely to continue to dominate.

4.2. Sensitivity Analysis

We examine below the scaleup, sizeup, and speedup characteristics of the Count distribution algorithm. We do not report further the results of the Data and Candidate distribution algorithms because of their inferior performance.

Scaleup. To see how well the Count distribution algorithm handles larger problem sets when more processors are available, we performed scaleup experiments where we increased the size of the database in direct proportion to the number of nodes in the system. We used the datasets *D2016K.T10.I2*, *D1456K.T15.I4* and *D1140K.T20.I6* from the previous experiments except that the number of transactions was increased or decreased depending upon the multiprocessor size. The database sizes for the single and 32 node configurations are shown in Table 1. At 100MB per node, all three datasets range from about 100MB in the single node case to almost 3.2GB in the 32 node case.

Figure 6 shows the performance results for the three datasets. In addition to the absolute response times as the number of processors is increased, we have also plotted scaleup which is the response time normalized with respect to the response time for a single processor. Clearly the Count algorithm scales very well, being able to keep the response time almost constant as the database and multiprocessor sizes increase. Slight increases in response times is due entirely to more processors being involved in communication. Since the itemsets found by the algorithm does not change as the database size is increased, the number of candidates whose support must be summed by the communication phase remains constant.

Sizeup. For these experiments, we fixed the size of the multiprocessor at 16 nodes while growing the database from 25 MB per node to 400 MB per node. We have plotted both the response times and sizeup in Figure 6. The sizeup is the response time normalized with respect to the response time for 25MB per node. The results show sublinear performance for the Count algorithm; the program is actually more efficient as the database size is increased. Since the results do not change as the database size increases neither does the amount or cost of communication. Increasing the size of the

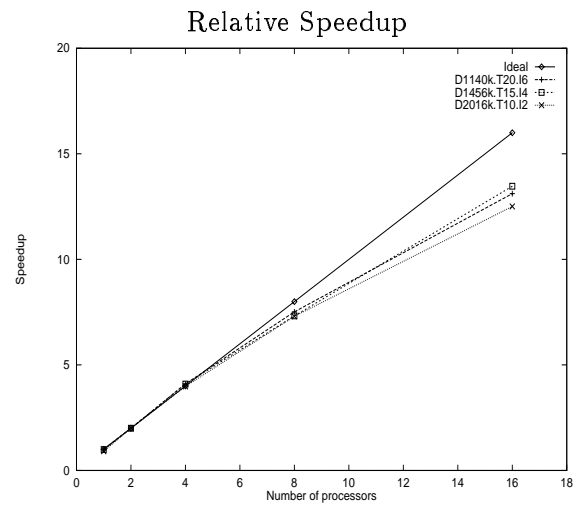
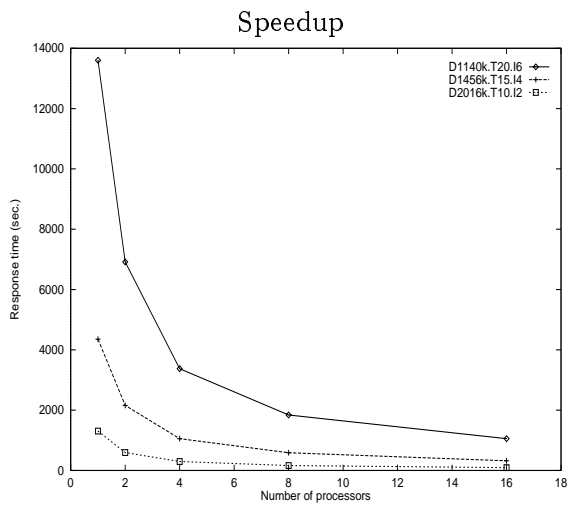
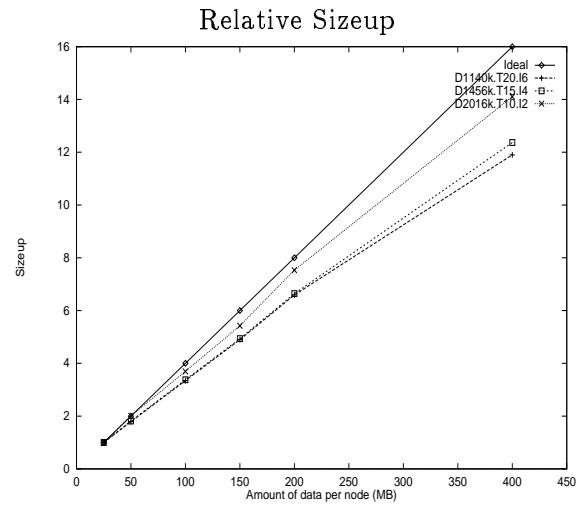
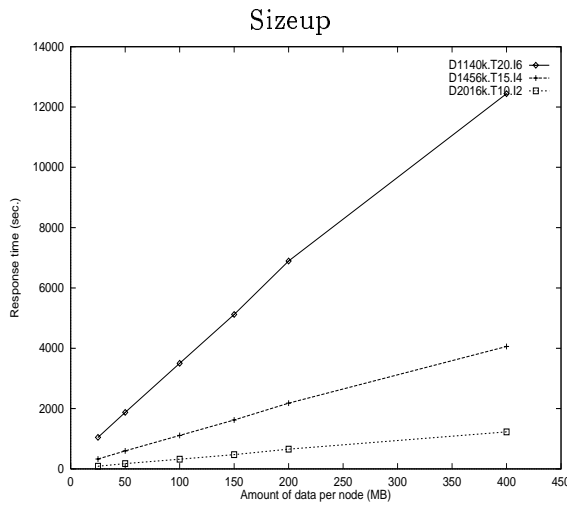
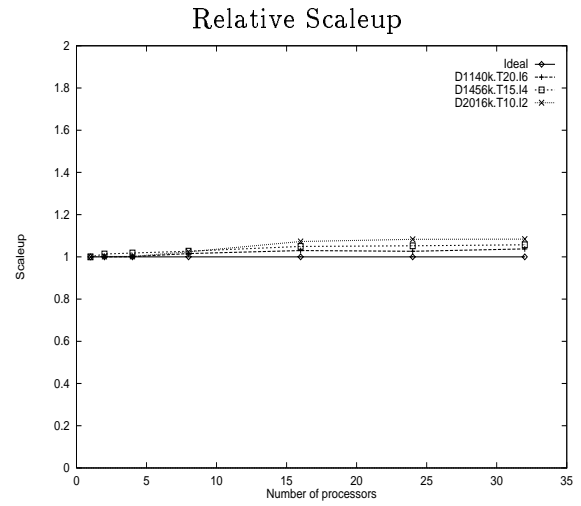
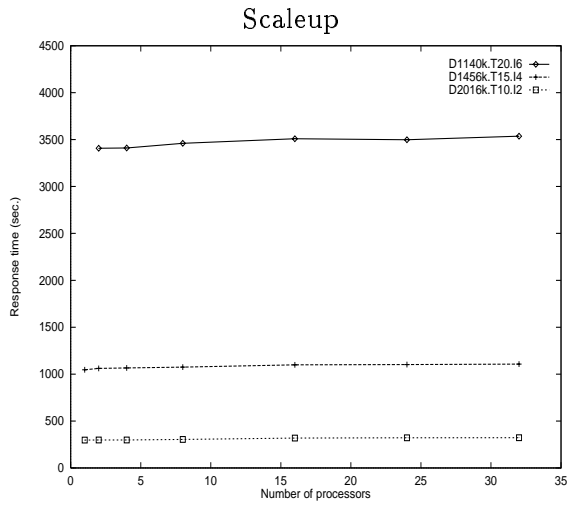


Figure 6: Performance of Count Distribution

database simply makes the non-communication portion of the code take more time due to more I/O and more transaction processing. This has the result of reducing the percentage of the overall time spent in communication. Since I/O and CPU processing scale perfectly with sizeup, we get sublinear performance.

Speedup. For our last set of experiments, we kept the database constant and varied the number of processors. Because of the constraint on available disk space, the size of each of the three databases was fixed at 400MB. Figure 6 shows the results of running the Count algorithm on configurations of up to 16 processors. We did not run with larger configurations because the amount of data at each node becomes too small. The speedup in this figure is the response time normalized with respect to the response time for a single processor. As the graphs show, Count has very good speedup performance. This performance does however begin to fall short of ideal at 8 processors. This is an artifact of the small amount of data each node processing. At only 25MB per node, communication times become a significant percentage of the overall response time. This is easily predicted from our sizeup experiments where we noticed that the more data a node processes, the less significant becomes the communication time giving us better performance. We are simply seeing the opposite effect here. Larger datasets would have shown even better speedup characteristics.

4.3. Effect of Hash Filtering

Recently, Park, Chen, and Yu [17] proposed the use of a hash filter to reduce the cost of Apriori, particularly in the second pass by reducing the size of C_2 . The basic idea is to build a hash filter as the tuples are read in the first pass. For every 2-itemset present in a tuple, a count is incremented in a corresponding hash bucket. Thus, at the end of the pass, we have an upperbound on the support count for every 2-itemset present in the database. When generating C_2 using L_1 , candidate itemsets are hashed, and any candidate whose support count in the hash table is less than the minimum support is deleted.

Figure 7 compares the combined response times for Pass 1 and 2 for the Count algorithm and this Hash Filter algorithm. The times for the remaining passes are identical. The Count algorithm beats Hash Filter because Count never explicitly forms C_2 ; rather, it uses a specialized version of the hash-tree as was done in [4]. Since nothing in C_2 can be pruned by the Apriori algorithm, it is equal to $L_1 \times L_1$. C_2 can thus be represented by a simple two-dimensional count array, drastically reducing memory requirements and function call overhead. Any savings from using the hash filter to prune C_2 are lost due to the cost of constructing the hash filter and the use a regular hash-tree for storing and counting C_2 .

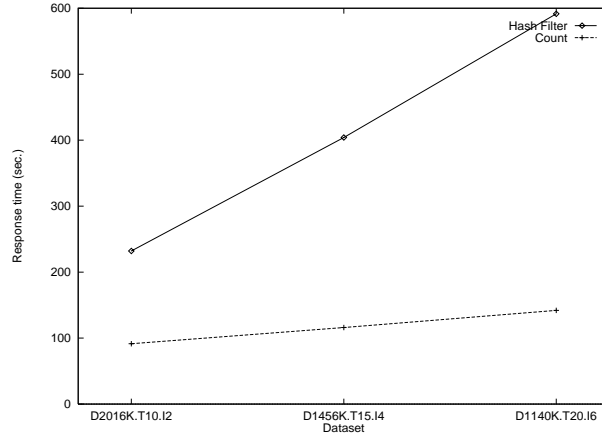


Figure 7: Effect of Hash Filtering

5. Conclusions

We considered the problem of mining association rules on a shared-nothing multiprocessor on which data has been partitioned across the nodes. We presented three parallel algorithms for this task. These algorithms represent a spectrum of trade-offs between computation, communication, memory usage, synchronization, and the use of problem-specific information.

The Count distribution algorithm was designed to minimize communication. No data tuples are exchanged between processors — only counts are exchanged. Processors can operate independently and asynchronously during the pass over the data, but need synchronization at the end of every pass.

The Data distribution algorithm was designed to exploit the aggregate memory of the system more effectively. In this algorithm, each processor counts mutually exclusive candidates. Thus, as the number of processors increases, a larger number of candidates can be counted in a pass. It can be effective if there are more candidates than what can fit in memory, forcing a pass into multiple subpasses when using the Count algorithm. The downside is that every processor must broadcast its local data to all other processors in every pass, making this algorithm viable only on a machine with very fast communication. Also, processors are still required to synchronize at the end of every pass in order to exchange frequent itemsets.

The Candidate distribution algorithm attempts to do away with the dependence between processors so that they may proceed independently without synchronizing. It tries to repartition the computation between processors in such a way that each processor can generate its set of candidates independent of all other processors. At the same time, the database is selectively replicated so that a processor can develop support counts for its candidates independent of all other processors. Thus, after repartitioning, each processor can proceed asynchronously. The Candidate distribution algorithm is the one that uses problem-specific semantics in its design.

We studied the above trade-offs and evaluated the relative performance of the three algorithms

by implementing them on 32-node SP2 parallel machine. The Count distribution emerged as the algorithm of choice. It exhibited linear scaleup and excellent speedup and sizeup behavior. When using N processors, the overhead was less than 7.5% compared to the response time of the serial algorithm executing over $1/N$ amount of data. The Data distribution algorithm lost out because of the cost of broadcasting local data from each processor to every other processor. The Candidate distribution algorithm did not win because the cost of data redistribution swamped the gains from not having to synchronize at the end of each pass.

Although we focussed on parallelizing the mining of association rules, the results and experience from this study have wider applicability. The implementation techniques we described for exchanging counts, distributing and repartitioning data, and redundant computations to save communications can be directly used in the design of parallel algorithms for other data mining tasks. Efficient counting is at the heart of several data mining algorithms [1]. Extrapolating from the results of this study, it is safe to conclude that one should focus (at least initially) on designs based on distributing counts for parallelizing these algorithms. Indeed that is how we are proceeding in parallelizing mining for multi-level association rules [10] [19], and sequential patterns [5]. Finally, we found the MPI (Message Passing Interface) communication primitives to be very powerful and convenient, and they simplified our code structure considerably. These primitives are part of a proposed message-passing communication standard [9], and they merit serious consideration in the design of parallel mining algorithms.

Acknowledgments. Maurice Houtsma implemented a parallel version of the associations mining algorithm presented in [2] on an earlier version of IBM POWERparallel System (called SP1) in which all nodes were diskless and all data were funneled through a master node. Although we could not use this implementation because of changes in architecture, communication library, and the basic algorithm, we benefitted from this experience. Howard Ho provided us the early prototype implementation of the MPI communication library to get us going. Ramakrishnan Srikant patiently explained many nuances of the serial Apriori implementation. Discussions with Mike Carey were influential in the initial stages of this work. Finally, several in the SP organization, particularly Bob Hieronymous, Sharon Selzo, and Bob Walkup, were wonderful in their help in arranging SP cycles to burn for our tests.

References.

- [1] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914–925, December 1993.

- [2] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining association rules between sets of items in large databases. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 207–216, Washington, D.C., May 1993.
- [3] Rakesh Agrawal, King-Ip Lin, Harpreet S. Sawhney, and Kyuseok Shim. Fast similarity search in the presence of noise, scaling, and translation in time-series databases. In *Proc. of the VLDB Conference*, Zurich, Switzerland, September 1995.
- [4] Rakesh Agrawal and Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules. In *Proc. of the 20th Int'l Conference on Very Large Databases*, Santiago, Chile, September 1994.
- [5] Rakesh Agrawal and Ramakrishnan Srikant. Mining Sequential Patterns. In *Proc. of the 11th Int'l Conference on Data Engineering*, Taipei, Taiwan, March 1995.
- [6] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The Gamma database machine project. In *IEEE Transactions on Knowledge and Data Engineering*, pages 44–62, March 1990.
- [7] Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. A database interface for clustering in large spatial databases. In *Int'l Conference on Knowledge Discovery in Databases and Data Mining (KDD-95)*, Montreal, Canada, August 1995.
- [8] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. In *Proc. of the ACM SIGMOD Conference on Management of Data*, May 1994.
- [9] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, May 1994.
- [10] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proc. of the 21st Int'l Conference on Very Large Databases*, Zurich, Switzerland, September 1995.
- [11] Jiawei Han, Yandong Cai, and Nick Cercone. Knowledge discovery in databases: An attribute oriented approach. In *Proc. of the VLDB Conference*, pages 547–559, Vancouver, British Columbia, Canada, 1992.
- [12] Maurice Houtsma and Arun Swami. Set-oriented mining of association rules. In *Int'l Conference on Data Engineering*, Taipei, Taiwan, March 1995.
- [13] Int'l Business Machines. *Scalable POWERparallel Systems*, GA23-2475-02 edition, February 1995.
- [14] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Efficient algorithms for discovering association rules. In *KDD-94: AAAI Workshop on Knowledge Discovery in Databases*, pages 181–192, Seattle, Washington, July 1994.

- [15] Manish Mehta, Rakesh Agrawal, and Jorma Rissanen. SLIQ: A fast scalable classifier for data mining. In *EDBT 96*, Avignon, France, March 1996.
- [16] Raymond T. Ng and Jiawei Han. Efficient and effective clustering methods for spatial data mining. In *Proc. of the VLDB Conference*, Santiago, Chile, September 1994.
- [17] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. An effective hash based algorithm for mining association rules. In *Proc. of the ACM-SIGMOD Conference on Management of Data*, San Jose, California, May 1995.
- [18] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. of the VLDB Conference*, Zurich, Switzerland, September 1995.
- [19] Ramakrishnan Srikant and Rakesh Agrawal. Mining Generalized Association Rules. In *Proc. of the 21st Int'l Conference on Very Large Databases*, Zurich, Switzerland, September 1995.
- [20] Teradata Corp. *DBC/1012 Data Base Computer System Manual*, C10-0001-02 release 2.0 edition, November 1985.

A. Partitioning Algorithm

We discuss in this appendix how the Candidate distribution algorithm partitions L_k . The partitioning strives to achieve the dual objective of allowing every processor to proceed as independently as possible and at the same time balancing the load on each of the processors.

Lemma 1. *Assuming that the items in the itemsets are lexicographically ordered, let \mathcal{E} be a set of frequent itemsets in L_k that have the same $k - 1$ long prefix (i.e., the first $k - 1$ items are the same). Let this prefix be l_{k-1} . Any candidate itemset c_m ($m > k$) that has l_{k-1} as the prefix is generated by the Apriori candidate generation procedure by recursively joining itemsets only in \mathcal{E} and the results thereof.*

PROOF. Immediate from the join step of the Apriori candidate generation procedure. \square

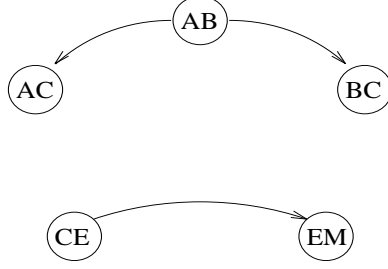
Example. Let L_3 be $\{ABC, ABD, ABE, ACD, ACE, BCD, BCE, BDE, CDE\}$. Then $L_4 = \{ABCD, ABCE, ABDE, ACDE, BCDE\}$, $L_5 = \{ABCDE\}$, and $L_6 = \emptyset$. Consider $\mathcal{E} = \{ABC, ABD, ABE\}$ whose members all have the common prefix AB . The candidates $ABCD$, $ABCE$, $ABDE$ and $ABCDE$ that also have the prefix AB are generated by joining only the items in \mathcal{E} . \square

Therefore, we can partition the itemsets in L_k based on common $k - 1$ long prefixes. By ensuring that no partition is assigned to more than one processor, we have ensured that each processor can generate candidates independently (ignoring the prune step). Suppose we repartition the database in such a way that any tuple that has any item contained in any of the L_k partitions assigned to a processor is also copied to the local disk of that processor. The processors then can proceed completely asynchronously.

This simple approach has two problems. A processor may have to obtain frequent itemsets computed by other processors for the prune step of the candidate generation. In the example above, the processor assigned the set \mathcal{E} has to know whether $BCDE$ is frequent to be able to decide whether to prune the set $ABCDE$, but the set with prefix BC may have been assigned to a different processor. The other problem is that we need to balance load across processors.

Lemma 2. *Let l_k be a frequent itemset in L_k . Consider a candidate itemset c_m ($m > k$), which is an extension of l_k . Let \mathcal{S} be the set of all itemsets of size $k - 1$ in l_k . To determine if c_m should be pruned, we need to consider only the frequent itemsets that are extensions of the itemsets in \mathcal{S} .*

PROOF. To decide on the pruning of c_m , we need to make sure that every subset of size $m - 1$ in c_m is frequent. Every such itemset is an extension of one of the $k - 1$ length subsets of l_k , and all these itemsets are in \mathcal{S} . \square



$$L_2 = \{ AB, AC, AM, AN, BC, GM, BN, CE, CM, \\ CN, CP, CQ, EM, EP, EQ, MN, MP, MQ, PQ \}$$

$$L_3 = \{ ABC, ABM, ABN, ACM, ACN, AMN, BCM, BCN, \\ BMN, CEM, CEP, CEQ, CPQ, EMP, EMQ, EPQ, MPQ \}$$

Figure 8: Example Dependency Graph

Example. Let l_k be $\{ABC\}$ and c_m be $\{ABCDEF\}$. Then $\mathcal{S} = \{AB, AC, BC\}$ and all $m - 1$ subsets of c_m are extensions of itemsets in \mathcal{S} . \square

Dependency Graph. Lemma 2 leads to the algorithm for constructing the dependency graph G that is used for candidate partitioning. This graph is created in pass k after computing L_k . There is a node in this graph for each itemset in \mathcal{S} for all l_k in L_k .

Let n be the node corresponding to $k - 1$ long prefix of l_k . If a $k - 1$ long prefix is such that it has only one k long extension in L_k , then no node is created for this itemset because it cannot generate any $k + 1$ long candidate. From n , we draw an arc to the nodes corresponding to all other $k - 1$ long itemsets in \mathcal{S} . This is done for each l_k in L_k that is an extension of the itemset corresponding to n . An arc from node n to node m represents that in order to decide the pruning of a candidate generated by extending the itemset corresponding to n , we need to consider extensions of the itemset corresponding to m (From Lemma 2).

Example. Consider the sets L_2 and L_3 given in Figure 8. We are interested in partitioning L_3 . Consider ABC . The set \mathcal{S} for ABC consists of $\{AB, AC, BC\}$. We create a node for AB and draw an arc from AB to AC and BC . Now consider ACM . The set \mathcal{S} for ACM consists of $\{AC, AM, CM\}$. But no out-going arc is created from AC because there is no node corresponding to AM or CM as they do not have more than one extension in L_3 . Similarly, for other itemsets in L_3 . \square

If load balancing was not a consideration, we could have assigned connected components of G to separate processors in a round-robin fashion and then each processor could have proceeded independently. To balance the load, every node n of G is assigned the weight:

$$\text{weight}(n) = \sum_{l_k} \text{support}(l_k)$$

where each l_k is an extension of the $k - 1$ long itemset corresponding to the node n . The weight of n is an estimate of the total number of transactions that will reside on the processor assigned this node after data repartitioning. It is a rough estimate since there is surely overlap in the sets of transactions that support each itemset. The weights are also an indicator of how many passes extensions of the itemsets will last; generally, the higher the support of an itemset, the more likely it is a subset of a much longer itemset.

An arc from node n to m is assigned the weight:

$$\text{weight}(n, m) = \text{support}(\text{itemset}(n) \cap \text{itemset}(m)) / \text{weight}(m).$$

Arc weights attempt to balance two concerns: data replication and useful pruning information. Since we would like to avoid replicating transactions onto more than one processor, we want to keep nodes that have many items in common together on the same processor. The numerator is a rough estimate of the number of transactions nodes n and m have in common and would have to be replicated if assigned to separate processors. The more transactions the nodes have in common, the more we would like to keep these two nodes on the same processor. The denominator estimates the usefulness of the information about node m in pruning future candidates of node n . If a subset of a candidate is not frequent, we gain by not having to count that candidate. However, if that subset is assigned to a different and slower processor, we may not receive information about that subset until after we have completed the prune step of the candidate generation. With no information available, we will be forced to assume that the subset is frequent and we will not be able to prune the candidate. So, if node n depends upon node m for future pruning, it would be best to assign the two nodes to the same processor if the weight of node m is low; if the weight is low, itemsets belonging to node m will probably not be extended much further before they are no longer frequent.

Partitioning. Nodes of the dependency graph are assigned to the processors in such a way that each processor has roughly equal total node weight. Having assigned a node to a processor, nodes connected with a higher weight arc are preferred for assigning to the same processor.

Let $W = \sum_n \text{weight}(n)$ where n is a node in G . Allocate a "bin" for each processor (effectively a set for holding itemsets). The weight each bin can hold is W/N , where N is the number of processors.

Find connected components of G . Calculate the total node weight of each component and sort them in order of decreasing weight.

Consider each connected component in decreasing order of weight and do the following for each component: Find the least loaded bin. If this bin can hold this component, assign it to the bin. Otherwise, assign the heaviest node of this component to the bin. Now, recursively find the heaviest edge from an assigned node to a non-assigned node, and assign the latter to the bin. This process stops when the bin is full. The remaining nodes of the original component now comprise one or more

smaller components. These are reinserted into the sorted list of unassigned components. This is repeated until no unassigned components remain.