Parallel Mining of Association Rules

Rakesh Agrawal *John C. Shafer*[*]

IBM Almaden Research Center
650 Harry Rd., San Jose, CA 95120
{ragrawal,shafer}@almaden.ibm.com
Tel: (408) 927-1734 Fax: (408) 927-3215

Abstract

We consider the problem of mining association rules on a shared-nothing multiprocessor. We present three algorithms that explore a spectrum of trade-offs between computation, communication, memory usage, synchronization, and the use of problem-specific information. The best algorithm exhibits near perfect scaleup behavior, yet requires only minimal overhead compared to the current best serial algorithm.

Keywords: data mining, association rules, sequential patterns, parallel algorithms

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.

Discovering association rules is an important data mining problem [1]. Recently, there has been considerable research in designing fast algorithms for this task [1] [3] [5] [6] [8] [12] [9] [11]. However, with the exception of [10], 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. In order to determine the best method for mining rules in parallel, we explore 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.

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

3. The *Candidate Distribution* algorithm exploits the semantics of the particular problem at hand both to reduce synchronization between the processors and to segment the database based upon the patterns the different transactions support. This algorithm also incorporates load balancing.

These algorithms are based upon the serial algorithm Apriori which was first presented in [3]. We chose the Apriori algorithm because of its superior performance over the earlier algorithms [1] [6], as shown in [3]. We preferred Apriori over AprioriHybrid, a somewhat faster algorithm in [3], because AprioriHybrid is harder to parallelize; the performance of AprioriHybrid is sensitive to heuristically determined parameters. Furthermore, the performance of Apriori can be made to approximate that of AprioriHybrid by combining the small workloads of several Apriori cycles into a single workload requiring only one cycle The algorithm in [8] is quite similar to Apriori and our parallelization techniques directly apply to this algorithm as well. The algorithm in [11] does not perform as well as Apriori on large datasets with a large number of items. The algorithm in [9] 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. Concurrent to our work, that algorithm has been parallelized and was recently presented with a simulation study in [10]. It too suffers from the use of a hash-filter, despite the use of a special communication operator to build it. We discuss this further in Section 4.3.

Our three parallel algorithms have all been implemented on an IBM POWERparallel System SP2 (henceforth referred to simply as SP2), a shared-nothing machine[7]. We present measurements from this implementation to evaluate the effectiveness of the design trade-offs. The winning algorithm is now part of the IBM data-mining product and is being used in the field.

The organization of the rest of the paper is as follows. Section 2 gives a brief review of the problem of mining association rules[1] and the Apriori algorithm[3] 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. A more detailed version of this paper can be found in [2].

2 Overview of the Serial Algorithm

2.1 Association Rules

The basic problem of finding association rules as introduced in [1] is as follows. Let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of literals, called items. Let $\mathcal{D}$ be a set of transactions, where each transaction T is an itemset such that $T \subseteq \mathcal{I}$. We say that a transaction T *contains* X, a set of some items in $\mathcal{I}$, if $X \subseteq T$. An *association rule* is an implication of the form $X \Rightarrow Y$, where $X \subset \mathcal{I}$, $Y \subset \mathcal{I}$, and $X \cap Y = \emptyset$. The rule $X \Rightarrow Y$ holds in the transaction set $\mathcal{D}$ with *confidence* c if $c\%$ of transactions in $\mathcal{D}$ that contain X also contain Y. The rule $X \Rightarrow Y$ has *support* s in the transaction set $\mathcal{D}$ if $s\%$ of transactions in $\mathcal{D}$ contain $X \cup Y$.

Given a set of transactions $\mathcal{D}$, the problem of mining association rules is to generate *all* association rules that have certain user-specified minimum support (called *minsup*) and confidence (called *minconf*).

Problem Decomposition The problem of mining association rules can be decomposed into two subproblems [1]:

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.

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 kth pass (there are k items in each candidate)

Figure 1: Notation

```
$L_1$ := {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

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 $conf =$ support($ABCD$)/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 [1] [3] [6] [8] [9] [11]. We review in Section 2.2 the apriori algorithm [3] on which our parallel algorithms are based.

2.2 Apriori Algorithm

Figure 2 gives an overview of the Apriori algorithm for finding all frequent itemsets, 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 [3] 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$_2$, ..., $p$.item$_{k-1}$, $q$.item$_{k-1}$
from $L_{k-1}$ $p$, $L_{k-1}$ $q$
where $p$.item$_1$ = $q$.item$_1$, ..., $p$.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 [4]. 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. Since each processor has the exact same C_k, each processor puts its count values in a common order into a count array. All that is needed now is to perform a parallel vector sum of the arrays. This only requires communicating count values and can be done in O(log(n)) communication steps. It also avoids any time-consuming logic that would otherwise be needed to assure that we only combine counts that belong to the same candidate. The full details of this process including the MPI communication primitives used are described in [2].

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 Count distribution algorithm counts no more candidates per pass than the serial algorithm.

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. On an N-processor configuration, Data will be able to count in a single pass a candidate set that would require N passes in Count. 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. 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.

The interesting step is Step 2 in which processors develop support counts for local candidates C_k^i asynchronously. During this step, processors are broadcasting their local data as well as receiving the local data of other processors. We must be careful to avoid network congestion and use asynchronous communication to overlap communication time with the counting of support. See [2] for full details.

3.3 Algorithm 3: Candidate Distribution

One limitation of both the Count and Data distribution algorithms is that since any database transaction could support any candidate itemset, each transaction must be compared against the entire candidate set. This is what requires Count to duplicate the candidate set on every processor and Data to broadcast every database transaction. Additionally, both Count and Data distribution algorithms require processors to synchronize at the end of each 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. These problems are due to the fact that neither Count nor Data exploit problem-specific knowledge; data tuples and candidate itemsets are partitioned merely to equally divide the work. All processors must be consulted and all information gathered before they can proceed onto the next pass.

The Candidate distribution algorithm attempts to do away with these dependencies by partitioning both the data and the candidates in such a way that each processor may proceed independently. 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 repartitioned so that a processor can count candidates in C_m^i independent of all other processors. Note that depending upon the quality of the itemset partitioning, parts of the database may have to be replicated on several processors. The itemset partitioning algorithm considers this aspect by identifying segments of L_{l-1} that are likely supported by different database transactions. 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, each processor proceeds independently, counting only its portion of the global candidate set using only local data. No communication of counts or data tuples is ever required. 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, this information is sent asynchronously, and processors do 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 instead 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.

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.

As in the Data distribution algorithm, Step 3 of pass $k = l$ requires communicating local data while support counts are being developed. The one difference here is that local data need not be broadcast to every other processor — because of the candidate partitioning, processors have some information about which transactions are useful in developing support counts on other processors. This allows processors to send less data through the network. Full details of this filtering are described in [2].

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 [2].

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 [3]. 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.

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.

3.5 Discussion of Tradeoffs

Initially, it was not clear to us which of our three algorithms would win, or if there would even be a single over-all winner. Count minimizes communication at the expense of ignoring aggregate memory. On a workstation-cluster environment, this approach is probably ideal; it may not be so, however, on an SP2. Data distribution which fully exploits aggregate memory at the cost of heavy communication will help us explore this issue. Also, Data's ability to count in a single pass N times as many candidates as Count could make this algorithm a strong contender. With the third algorithm, Candidate distribution, we will see if incorporating detailed problem-knowledge can yield the benefits of both the Count and Data distribution algorithms. We will also see how beneficial removing processor dependence and synchronous communication can be.

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 [7] 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
D2016K.T10.I4	10	4	2016K	32256K	64512K
D1456K.T15.I4	15	4	1456K	23296K	46592K
D1140K.T20.I4	20	4	1140K	18240K	36480K
D1140K.T20.I6	20	6	1140K	18240K	36480K

T	Average transaction length
I	Average size of frequent itemsets
$\mathcal{D}$	Average number of transactions

Table 1: Data Parameters

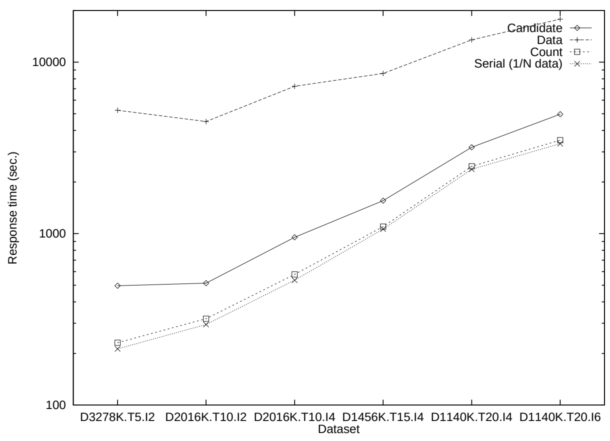
We used synthetic datasets of varying complexity, generated using the procedure described in [3]. The characteristics of the six datasets we used are shown in Table 1. These datasets vary from many short transactions with few frequent itemsets, to fewer larger transactions with many frequent itemsets. All the datasets were about 100MB per processor in size. We could not use larger datasets 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 3 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. One third of that overhead, about 2.5%, was spend waiting for the 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



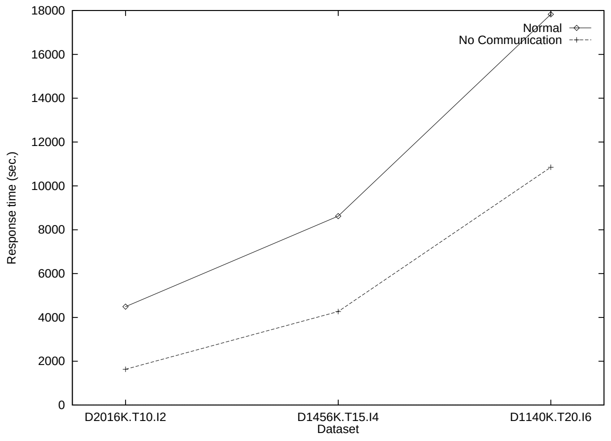


Figure 3: Relative Performance of the Algorithms

Figure 4: Communication Costs for Data Distribution

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

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 transaction. Communication is the worst of these two problems as show by Figure 4, 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. The algorithm was also almost entirely CPU-bound, making I/O savings due to Data making fewer passes practically negligible.

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 completely independently with smaller candidate sets can compensate for this cost. As the performance results show, redistribution simply costs too much.

Also, unlike Data distribution, the Candidate algorithm was unable to capitalize on its more optimal use of aggregate memory; the large candidate sets that force Count into multiple subpasses all occur before Candidate takes over with its redistribution pass. Candidate thus makes just as many data passes as Count. These insufficient gains coupled with a high redistribution cost allow Count with its small overhead to emerge as the overall winner.

Although our experiments show Count's overhead to be fairly small, 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 5 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 5. 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 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.

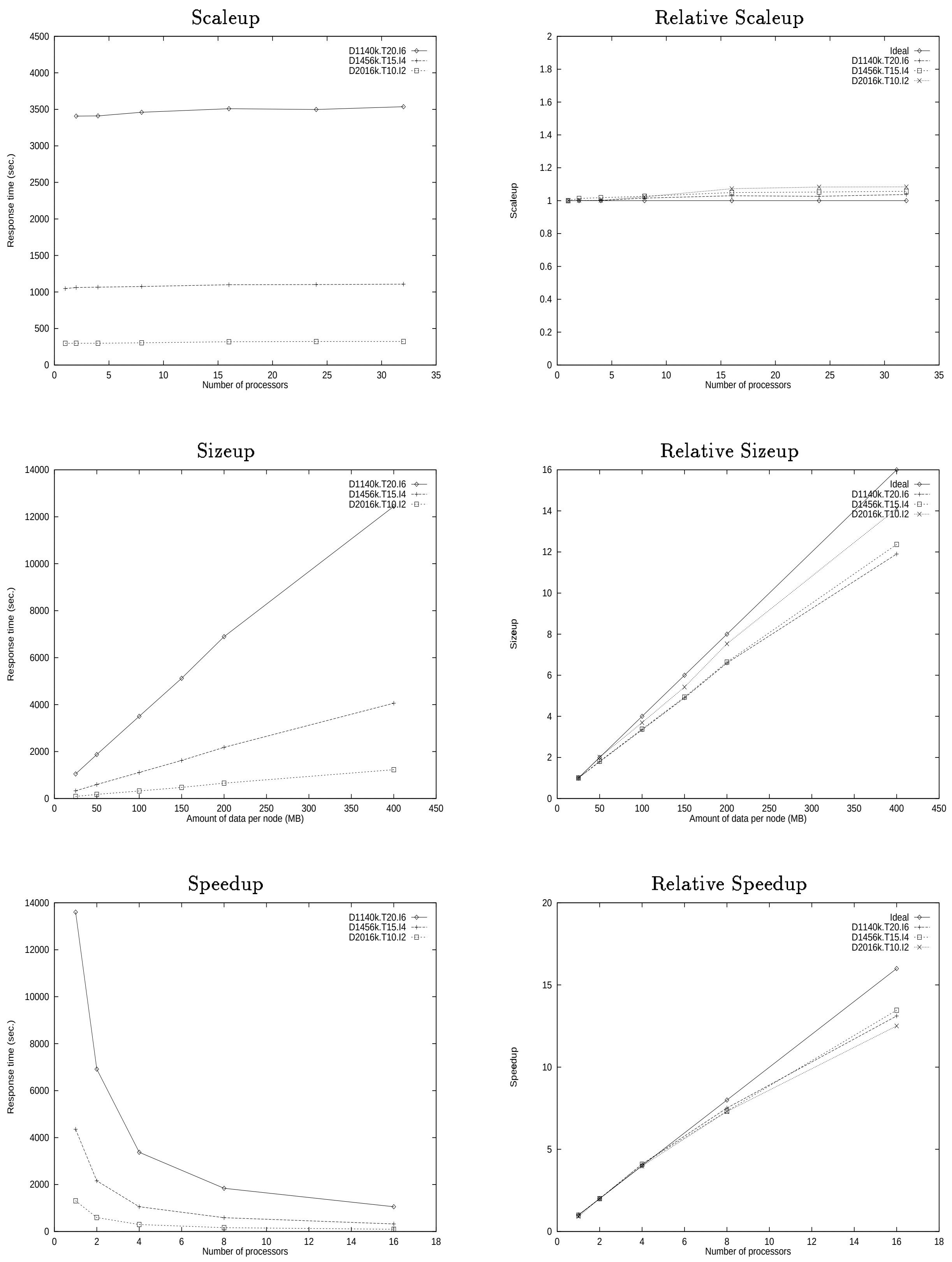


Figure 5: Performance of Count Distribution

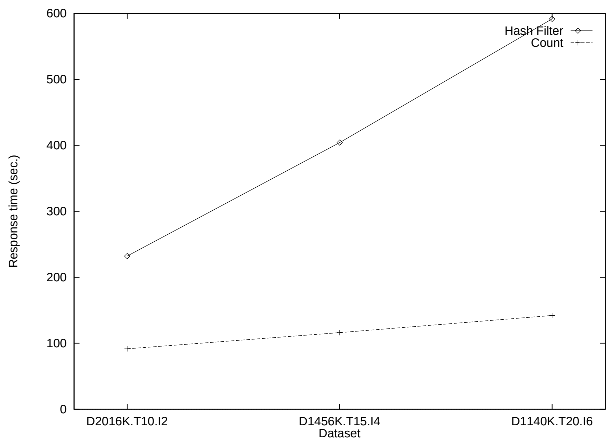


Figure 6: Effect of Hash Filtering

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 5 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 [9] 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 6 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 [3]. Since nothing in C_2 can be pruned by the Apriori candidate generation 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.

A parallel version of this Hash Filter algorithm called *PDM* has been presented in [10], along with performance results from a simulation study. It uses a parallelization technique similar to that of Count, except that entire candidate sets are exchanged rather than just the candidate counts. This is more expensive in both communication and CPU costs. The focus in *PDM* was on the efficient construction of the same hash-filter used by the serial algorithm to speed up pass two. However, as in the serial algorithm, the hash-filter actually hurts performance, resulting in a double performance hit in *PDM*.

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 based upon Apriori, the best serial algorithms for mining association rules. The designs of these algorithms represent a spectrum of trade-offs between computation, communication, memory usage, synchronization, and the use of problem-specific information.

The Count distribution algorithms attempts to minimize communication by replicating the candidate sets in each processor's memory. Processors work only with local data and only communicate counts. The Data distribution algorithm takes the counter approach where each processor works with the entire dataset but only portion of the candidate set. This maximizes the use of aggregate memory, but requires high communication to broadcast all the data. Again, while minimizing communication may be the best approach for a workstation-cluster environment, this is not necessarily true for an SP2. Lastly, the Candidate algorithm incorporates domain-knowledge to partition both the data and the candidates, allowing each processor to work on a unique set of candidates without having to repeatedly broadcast the entire dataset. This maximizes the use of aggregate memory while limiting heavy communication to a single redistribution pass. This also completely eliminates the synchronization costs that Count and Data must pay at the end of every pass.

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. Our results show that even on a high-bandwidth/low-latency system such as an SP2, data redistribution is still too costly.

The Candidate distribution algorithm is similarly edged out because of the cost of data redistribution; gains from having each processor work independently on a different subset of the problem could not make up for single pass of redistribution. While it may be disheartening to learn that a carefully designed algorithm such as Candidate can be beaten by a relatively simpler algorithm like Count, it does at least illuminate the fact not all problems require an intricate parallelization. By exploring the various possibilities, we have shown that this is true for mining association rules.

Acknowledgments Maurice Houtsma implemented a parallel version of the association-rule mining algorithm presented in [1] 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 benefited 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 for our tests.

References

[1] 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.

[2] Rakesh Agrawal and John Shafer. Parallel mining of association rules: Design, implementation and experience. Research Report RJ 10004, IBM Almaden Research Center, San Jose, California, February 1996. Available from `http://www.almaden.ibm.com/cs/quest`.

[3] 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.

[4] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, May 1994.

[5] 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.

[6] Maurice Houtsma and Arun Swami. Set-oriented mining of association rules. In *Int'l Conference on Data Engineering*, Taipei, Taiwan, March 1995.

[7] Int'l Business Machines. *Scalable POWERparallel Systems*, GA23-2475-02 edition, February 1995.

[8] 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.

[9] 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.

[10] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. Efficient parallel data mining for association rules. In *Fourth Int'l Conference on Information and Knowledge Management*, Baltimore, Maryland, November 1995.

[11] 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.

[12] 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.