D-007

Tree-based Parallel Algorithms with Reduced Inter-Processor Communication for Association Rule Mining

NGUYEN VIET ANH[†], SHIGERU OYANAGI[†], AND KATSUHIRO YAMAZAKI[†]

1 Introduction

In this paper we present two parallel algorithms for mining association rules that are well suited for distributed memory parallel computers. The algorithms are developed based on FP-growth method [3]. The first algorithm is a task parallel formulation using a static load balancing technique. The second algorithm improves upon the first algorithm by dynamically balancing the load when the static task assignment leads to load imbalance. We use the count matrix technique to compute the weight of tasks and to distribute tasks to processors. This technique also helps to reduce the time needed to scan the trees and to reduce communication cost. We also use the hash tree technique to group similar prefix-paths extracted from the tree, and thus can greatly reduce the amount of information exchanged among processors. Our experiments show that the algorithms are capable of achieving very good speedups, and of substantially reducing the amount of time when finding frequent patterns in very large databases.

2 Initial database partitioning

If p is the total number of processors, the original database is initially partitioned into p equal-size parts, each one then assigned to different processor. Each processor scans the database once and enumerates the local occurrences. All processors then obtain frequent items by exchanging local counts with all other processors using a global sum-reduction operation. These frequent items are sorted in support descending order to form L, list of frequent items. Note that L is identical for all processors. Each processor scans its local data to build the local FP-tree in exactly the same way with serial FP-tree algorithm [3].

ID	Items	(Ordered) Freq Items	Processor
1	A B C D E	B A D	P_0
2	F B D E G	B D F G	
3	B D A E G	B A D G	
4	A B F G D	B A D F G	P_1
5	B F D G K	B D F G	
6	A B F G D	B A D F G	
7	A R M K O	A	P_2
8	B F G A D	B A D F G	
9	A B F M O	B A F	

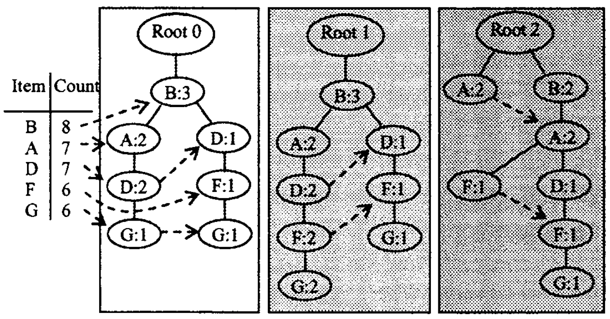

Figure 1. Database partitioning and local FP-trees

To understand this process, let's examine the example in Figure 1. Suppose we have 3 processors. The transactional database is described in the table. The minimum support in this example is set to 5.

Building the local FP-trees is not a final goal but a means to discover all frequent patterns without any additional database scan. According to FP-growth method, in order to mine all frequent patterns concerning item i (in header table) we need to construct i's conditional pattern base and then i's conditional FP-tree T_i. The branches that contain item i may reside in multiple local FP-trees, and we need to collect them from all possessing processors to form i's conditional base. We do not prove the correctness of this approach because of the length limitation of this paper but focus on how to distribute the works evenly among all processors and reduce the work load imbalance.

3 Static Load Balancing Algorithm (SLB)

The key idea behind this algorithm is to calculate the amount of computation for each item in the header table of the first FP-tree (the tree constructed from the original database) and then to divide those items into p equal parts and assign each part to a processor. When processing item i we first build i's conditional pattern tree and recursively generate subsequent conditional pattern bases and subsequent FP-trees until all frequent patterns are mined. The number of iterations is exponentially proportional to the height of i's conditional FP-tree that is bounded by number of items in its header table. Therefore, the length of the header table of the conditional FP-tree of item i can be used to measure i's computation cost.

In FP-growth method, for each item i in the header of the FP-tree $T_\emptyset$, two traversals of $T_\emptyset$ are needed for constructing the conditional FP-tree T_i. The first traversal constructs header table of the new tree by finding all frequent items in the conditional pattern base of i. The second traversal constructs the new tree T_i. In parallel environment, after scanning the trees, all processors need to communicate with each other to form the global information. Note that the support for an item j in the conditional pattern base of i is, in fact, the number of transactions that contain both j and i. If we have an array that store the count of all pair items, we can get the header table directly from that count array, and therefore, the first tree scan and information exchanging can be omited. Figure 2 shows how the count matrix is used to calculate the weight of item G.

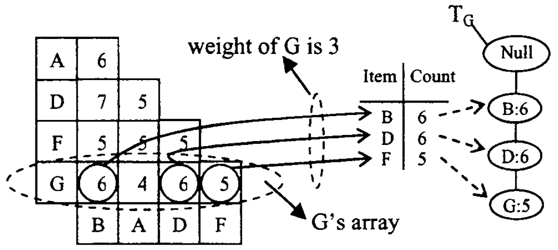

Figure 2. Using count matrix to measure item weight

After having the computation weight of all items, we use the bin-packing algorithm to pack them into p equal-sized buckets. Each bucket is then assigned to one processor.

The count matrix contains the count arrays of all items in database. It is stored in a file and is loaded each time the pro-

[†]Graduate School of Science and Engineering, Ritsumeikan University

gram runs. Only count arrays for frequent items are extracted by simply reading first n-1 rows of the file into allocated arrays, with n is the number of frequent items. The count matrix is built in a preprocessing step and is maintained during the life time of the database. By using count matrix we can also reduce the time for second Fp-tree scan because we can neglect all infrequent items when examining the paths from nodes in node-links up to the root of the tree.

Now let us examine the conditional bases exchanging issue. For an item in the header table, millions of prefix-paths can be found from the tree, and thus lead to a huge amount of information needed to be exchanged among processors. However, in fact, paths with the form $i_1 \rightarrow i_2 \rightarrow \ldots \rightarrow i_n$ may appear many times, and only differ in the support counts. What we need to do is to sum up all the counts and send it with the path $i_1 \rightarrow i_2 \rightarrow \ldots \rightarrow i_n$ only once. In our algorithm this problem is solved by using hash tree technique.

4 Dynamic Load Balancing Algorithm (DLB)

This algorithm monitors the load of processors and redistributes the work between processors when the static task assignment leads to the load imbalance.

The algorithm works as follows. The early period of this algorithm is similar to SLB. However, processors will not process independently until all patterns are mined. When a processor finishes its portion of allocated work, it selects a donor processor and sends it a request for work. If the donor processor does not contain enough work, it will send a rejection; otherwise it will send part of its work to requesting processor. Upon receiving new work the processor starts processing newly received work until it becomes idle again. A control process is used to maintain the rank of the process to which the next request will be sent, and when a process run out of work, it get the rank from the control process. This process continues until every processor completely processed all items assigned to it. Each processor manages a local stack whose stack node contains an item and its corresponding conditional pattern base. Dijkstra-Scholten termination detection algorithm [2] is used to detect whether all processors finished their work and there was no message in transition, and thus the overall computation have finished.

One key issue in the dynamic load balancing approach is managing the task granularities. If the weight of this task is too small, the amount of communication may increase. Otherwise, if the weight of this task is too large, it will take a significant amount of time before a processor can service work requests. In both cases the overall performance can be decreased. To deal with this problem we use a parameter named *maxload*. When processing an item with the weight greater than *maxload*, if a subsequent conditional pattern base has the weight smaller than *maxload*, it will be processed until completion. All subsequent conditional pattern bases having weight greater than *maxload* will be put into local stack and will be processed in the next iterations.

We have carried out several optimizations to increase the performance of the algorithm, namely, the control of servicing time, the order of items to be processed, and avoiding blocking when exchanging conditional pattern bases.

5 Experimental Results

All experiments were performed on the PC Cluster that consists of 16 nodes. Each node has 2 processors Intel Xeon 2.8GHz, 2 GB of memory, and 80 GB of hard disk. The nodes are connected by 1Gbs Ethernet network. For performance evaluation we use different synthetic datasets generated using procedure described in [1].

Figure 3 compares the scalability of SLB and DLB as the minimum support decreases from 0.2% to 0.08%. The experiments are performed on an 8 node configuration on two datasets, T25I10D2000K and T25I20D1000K.

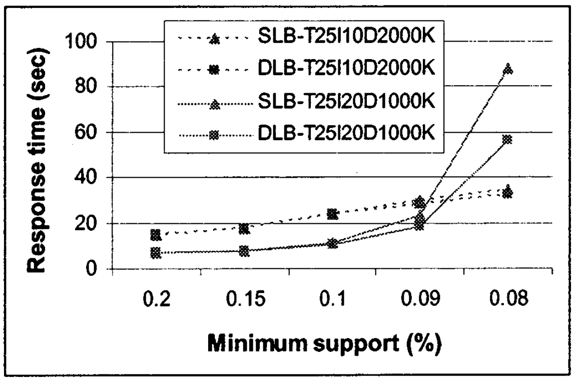

Figure 3: Scalability with minimum support

SLB scales almost equally with DLB on T25I10D2000K. However, DLB scales much better than SLB on T25I20D1000K. This is because with T25I20D1000K, when the minimum support goes down, the number of long frequent itemsets increase dramatically compared with dataset T25I10D2000K, and thus likely lead to the load imbalance.

Figure 4 shows the speedups obtained for two algorithms on the dataset T25I20D1000K when minimum support is set to 0.1%. Both SLB and DLB have very good speedup performance. When the number of processors increases, DLB run a little bit better than SLB. This is because the load imbalance tends to increase with bigger numbers of processors.

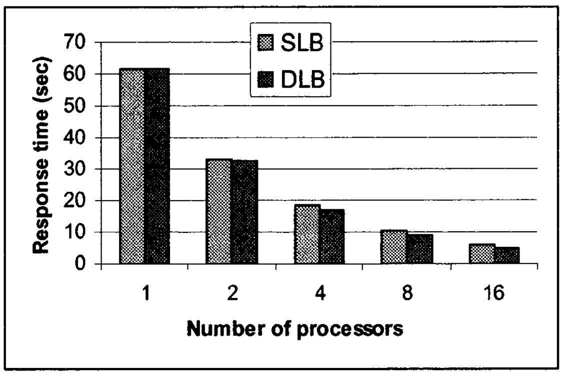

Figure 4: Speedup

6 Conclusion and future works

We propose two parallel algorithms for mining association rules based on FP-growth method. In many situations, the algorithms can reduce the communication required to exchange the conditional bases. Our experiments confirms that both SLB and DLB achieve very good performance and good speedup in large databases. Our future research will explore the parallel aspects of these algorithms in more details.

References

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

[2] E. W. Dijkstra, W. H. Seijen, and A. J. M. Van Gasteren. *Derivation of a termination detection algorithm for a distributed computation.* Information Processing Letters, 1983

[3] J. Han, J. Pei, and Y. Yin. *Mining frequent patterns without candidate generation.* In Proc. of the ACM SIGMOD Conf. on Management of Data, 2000