Hmine-rev: H-mine 並列化に向けて大容量データベースにおける頻出パターンマイニングに関する研究

ボウォ プラスティョ
praz@tkl.iis.u-tokyo.ac.jp
東京大学

イコ プラムディオノ
iko.pramudiono@lab.ntt.co.jp
日本電信電話株式会社
NTT 情報流通プラットフォーム研究所

喜連川 優
kitsure@tkl.iis.u-tokyo.ac.jp
東京大学生産研技術究所

H-mine はスパースなデータセットに対して高性能と知られているメモリベースマイニングアルゴリズムであるが、動的 H-struct リンク調整という H-mine 特有の処理は並列化を困難にする。本論文では H-struct リンク調整を一切必要としない改善されたアルゴリズムを提案する。提案アルゴリズムは、オリジナルバージョンと匹敵する性能を持ちながら、並列処理に容易に拡張することが可能となる。

Hmine-rev: Toward H-mine Parallelization on Mining Frequent Patterns in Large Databases

Bowo Prasetyo
praz@tkl.iis.u-tokyo.ac.jp
University of Tokyo

Iko Pramudiono
iko.pramudiono@lab.ntt.co.jp
NTT Information Sharing Platform Laboratories
NTT Corporation

Masaru Kitsuregawa
kitsure@tkl.iis.u-tokyo.ac.jp
Institute of Industrial Science, University of Tokyo

H-mine is a frequent pattern mining algorithm that takes advantage of a hyper-linked H-struct data structure, runs fast in memory-based setting, and is known to have high performance in a sparse data set. However, H-mine's inherent necessity to dynamically adjust H-struct links in the middle of mining process makes it difficult to do any parallelization effort on the algorithm. In this study, we propose a revised algorithm of H-mine that does not need any adjustment of H-struct links by modifying link structure and reversing the order of processing data. The revised algorithm has comparable performance with the original version and can be easily extended to use in parallel environment.

1. Introduction

One of the most researched field in the data mining area is frequent pattern mining algorithm due to difficulties of doing it efficiently. Apriori algorithms [1] that can significantly reduce mining time by generating pattern candidates had successfully brought many researchers attention. Then, FP-growth algorithm [2] for dense data set and H-mine algorithm [3] for sparse data set that scans data set only twice by utilizing in-memory data structures had changed the main stream of frequent pattern mining algorithm from Apriori-like to pattern-growth algorithm.

Today, parallel computing technology provides

many possibilities to do frequent pattern mining more efficiently by combining many CPUs power to work together. For example, there is already a parallel version [4] of FP-growth algorithm that is known to work efficiently for dense data set. However, H-mine algorithm that is known to work efficiently on sparse data set has no parallel version up to now due to its inherent problem, i.e. H-struct link adjustment during the middle of mining process.

In this research we have developed a modified version of H-mine that can be easily extended to its parallel version. First we modify H-struct link structure and reverse the order to process data set, and we called the new algorithm "Hmine-rev" (reversed H-mine). Next we apply parallelization techniques to Hmine-rev algorithm using MPICH [5] as parallelization interface, and performed some experiments on artificial data set to test the performance of new algorithm and its parallel version.

In section 2 related works on the topic will be described briefly. Section 3 and 4 will be dedicated to explain H-mine algorithm and its difficulties to do parallelization. In section 5 and 6 we will explain Hmine-rev algorithm and its parallelization effort. Section 7 shows the experimental results and section 8 describes the conclusions and future works.

2. Related Works

In 2001 J. Pei et. al [3] proposed an algorithm to mine frequent patterns efficiently on a sparse data set. This algorithm utilizes H-struct data structure that has very limited and predictable space overhead, and runs very fast in memory setting. We describe H-mine in detail in chapter 3 in order to understand our proposed algorithm that modifies the algorithm.

P. Iko et. al. in 2003 [4] proposed a parallel version of FP-growth algorithm that is known to work very efficient on a dense data set. The parallel algorithm makes use the property in FP-growth that is processing of a conditional pattern base is independent from other conditional pattern base. Utilizing this property the algorithm dynamically assigns which node should process which conditional pattern base in parallel.

3. H-mine Algorithm

Table 1. Data set, fitems: *a,c,d,e,g*.

Trans ID	Items	Fitem projection
100	*c,d,e,f,g,i*	*c,d,e,g*
200	*a,c,d,e,m*	*a,c,d,e*
300	*a,b,d,e,g,k*	*a,d,e,g*
400	*a,c,d,h*	*a,c,d*

For example, we have a data set as shown in Table 1 below, and want to find all frequent patterns that satisfies minimum support of 2. H-mine algorithm to find frequent patterns can be divided in two phases, *building* data structures that involved two times scanning of data set and *mining* the in-memory data structures built.

3.1. Building the data structures

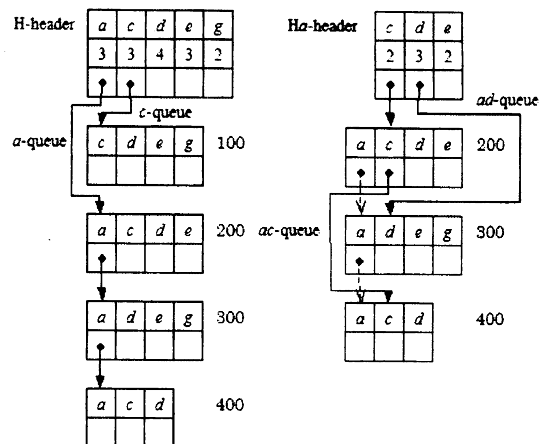

Figure 1. H-header with H-struct (left) and Ha-header (right)

1. Scan the data set once for all transactions and items to find all frequent items (fitem) and store it in an array of data structure called *H-header* (Figure 1 left) ordered alphabetically by item. H-header has three data slots, *fitem*, its *counts* and *a link* to H-struct. Fitems found are *a,c,d,e,g*. At this stage we have already found five frequent patterns, those are fitems

itself in H-header.

2. Scan the data set once more to project all transactions into fitems and store it in arrays of data structure called *H-struct* (Figure 1 left). H-struct has two data slots, *item* and *a link* to next H-struct. In this stage, the initial H-header and H-struct links (level 1 queues) that connect all the same items in the *first position*, are built. The rest of mining is then performed on these in-memory data structures.

3.2. Mining the data structures

Mining process in H-mine must be done in alphabetical order of frequent items as stated in H-header: *a-c-d-e-g*. For example, first we will mine all patterns containing '*a*', then all patterns containing '*c*' (without '*a*'), and so on.

To find all patterns containing '*a*':

1. From H-header follow level 1 *a*-queue (the link of item *a*), find all frequent items appearing after '*a*' and store it in conditional H-header called H*a*-header (Figure 1 right). In the same time, build the next links in H-struct (level 2 queues), i.e. *ac*-queue and *ad*-queue that connect all the same items following '*a*'. Here we found another three frequent patterns, '*ac*', '*ad*', '*ae*' as stated in H*a*-header.
2. Similarly, from H*a*-header follow level 2 *ac*-queue and find all frequent items appearing after '*c*' and store it in conditional H-header called H*ac*-header (Figure 2 left). In the same time, build the next links in H-struct (level 3 queues), i.e. *acd*-queue that connects all the same items following '*ac*'. Here we found another frequent pattern, '*acd*' as stated in H*ac*-header.
3. Repeat this process recursively until there is no more level of queue can be built, and then return to the previous level of H-header and continue to mine the next queue for all queues.
4. When the algorithm return to the previous level of H-header it needs to adjust some links in H-struct that has been built so far. For example, because there is no more level of queue can be built after '*acd*', algorithm will return to the previous level of H-header (e.g. H*a*-header) and continue to mine the next queue, *ad*-queue. Before continuing mining process, the algorithm needs to insert *acd*-queue into *ad*-queue to get the full *ad*-queue in H-struct (Figure 2 right).

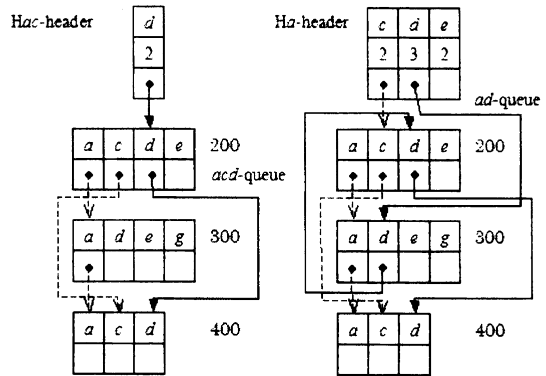

Figure 2. H*ac*-header (left) and Adjusted ad-queue (right)

4. H-mine Parallelization Problem

When trying to parallel H-mine one will soon realizes that there is a problem inherent to the algorithm itself which greatly increases the difficulties of parallelization efforts. The problem is H-struct link adjustment that is performed dynamically in the middle of mining process, i.e. when returning to the previous level of H-header after finished mining a queue. Therefore, the algorithm cannot proceed to mine the next queue before finishing the current queue to get information to adjust the links.

Link adjustment is absolutely required in the algorithm in order to mine all frequent patterns from data set completely, else the mined patterns will be incomplete. Thus the mining process in each queue of H-mine is dependent on its previous queue. This property makes parallelization is impossible unless we can take away this queue dependency from the algorithm.

5. Hmine-rev Algorithm

Hmine-rev differs from H-mine algorithm in two aspects: link structure and data processing order.

Hmine-rev's link structure is designed in such a way that it does not need any kinds of link adjustment. And the data is processed in the reversed way compared to that of H-mine algorithm. This two modifications has successfully removed any queue dependencies in the algorithm, and thus made it possible to do parallelization.

The algorithm of Hmine-rev in general is similar to that of H-mine, the complexity and performance is also about the same order. Again we will use data set in Table 1 above for example, and find all frequent patterns that satisfies minimum support of 2. Similar to H-mine, the algorithm of Hmine-rev can be divided in two phases, *building* data structures that involved two times scanning of data set and *mining* the data structures built in memory.

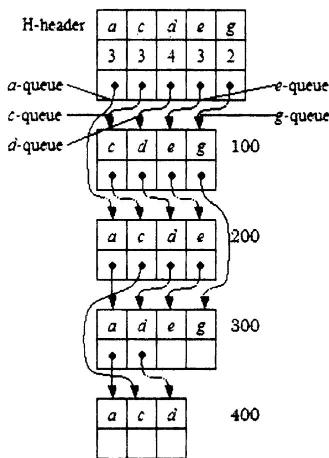

Figure 3. H-header and H-struct (Hmine-rev)

5.1. Building the data structures

1. Scan the data set once for all transactions and items to find all frequent items and store it in an array of *H-header* (same as H-mine) ordered alphabetically by item. Fitems found are *a,c,d,e,g*. At this stage we have already found five frequent patterns, those are fitems itself in H-header.
2. Scan the data set once more to project all transactions into fitems and store it in arrays of *H-struct* (Figure 3). In this stage, the initial H-header and H-struct links are built. Note that differ to H-mine, initial link structure in Hmine-rev are built completely for *all queues*, not only for items in the first position. The rest of mining is then performed on these data structures.

5.2. Mining the data structures

Differ to H-mine, data mining process in Hmine-rev algorithm must be done in reversed alphabetical order of frequent items as stated in H-header: *a-c-d-e-g* (from backward). For example, first we will mine all patterns containing '*g*' (without *a,c,d,e*), then all patterns containing '*e*' (without *a,c,d*), and so on.

Finding all patterns containing '*g*' (without *a,c,d,e*) is not needed, because we have already found them just as stated in H-header (2 patterns). To find all patterns containing '*e*' (without *a,c,d*): from H-header follow *e*-queue, find all frequent items appearing after '*e*' and store it in conditional H-header called H*e*-header (Figure 4 right). Here we found a frequent pattern, '*eg*' as stated in H*e*-header.

To find all patterns containing '*d*' (without *a,c*):

1. From H-header follow *d*-queue, find all frequent items appearing after '*d*' and store it in conditional H-header called H*d*-header (Figure 4 left). Here we found another two frequent patterns, '*de*' and '*dg*' as stated in H*d*-header.
2. Similarly, from H*d*-header follow *dg*-queue and find all frequent items appearing after '*g*' and we found nothing. So we go to the next queue '*de*', find all frequent items appearing after '*e*' and store it in conditional H-header called H*de*-header (Figure 4 right). As counting items we link all same items together and build new links in H-struct for the next level queues. Here we found another frequent pattern, '*deg*' as stated in H*de*-header. Note that when we move from *dg*-queue to *de*-queue, we do nothing to the H-struct link structure. There is *no link adjustment* between queues as required in H-mine.

3. Repeat this process recursively until there is no more level of queue can be built, and then return to the previous H-header and continue to mine the next queue for all queues.

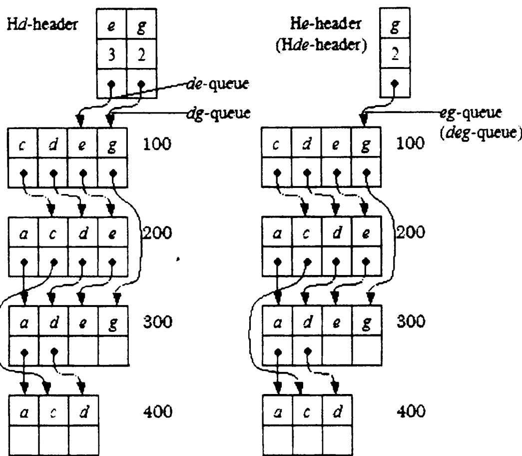

Figure 4. H*d*-header (left) and H*e*-header/H*de*-header (right)

Note that we must perform mining process in the reversed alphabetical order to get proper results. The reason is that in the step 2 above we are building new links in H-struct while counting items in each queue. This process alters link structure for queues in the next *proper alphabetical order*. Therefore, we need to do mining process in reversed alphabetical order, such that link structure altered in the process will belong to the queues that is already mined.

6. Hmine-rev Parallelization

Since there is no link adjustment in Hmine-rev algorithm, the parallelization is possible and quite straight-forward. Here we will describe Hmine-rev parallelization techniques on shared-disk and shared-nothing environment.

6.1. Pre-assigned Parallelization in Shared-Disk / SAN Environment

An example of a shared-disk parallel environment is a PC cluster that consists of several nodes of CPU that are connected to the same logical storage device or storage area network (SAN). One way to do parallelization in this environment is by pre-assigning in advance each level 1 queue to different nodes as shown in Figure 5.

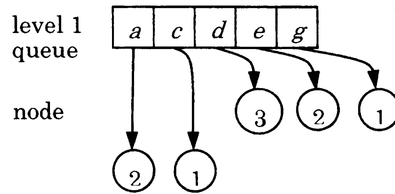

Figure 5. Pre-assigned parallelization

The advantage of pre-assigning every level 1 queue to every node in advance is that there is no need to check for available idle nodes during the mining and thus may avoid any communications between each node. Moreover, since Hmine-rev has no link adjustment, each node can mine its own assigned queue independently without requiring any data exchanges in the middle of mining. Therefore, this parallelization technique is ideal and the most efficient method possible regarding the number of data exchange and inter-node communication.

Algorithm of pre-assigning technique is:

1. Each node builds H-header and H-struct in local memory using the same data from shared disk. The constructed H-header and H-struct in all nodes should be identical.
2. Assign each level 1 queue to different nodes in advance (Figure 5). For example, *g* and *c*-queue are assigned to node 1, *e* and *a*-queue to node 2, and *d*-queue to node 3. Each node then perform data mining on assigned queue independently without even doing any data-exchange nor communication during the mining process. After completing mining process, each node writes its results separately to the shared disk.

6.2. Pre-assigned Paralellization in Shared-Nothing Environment

Real-world distributed system is often sharing

nothing between each node. It is usually a group of several computers, each having its own CPU, memory and hard disk with its own data and connected together via a network cable. Therefore, it is needed to build a parallelization algorithm that is capable to process data set that is distributed in several computers.

Below is a parallel algorithm to process a data set that is distributed in several nodes.

1. Each node builds local H-header and H-struct using its own local data and then exchanges to each other to build global H-header and H-struct (Figure 6).

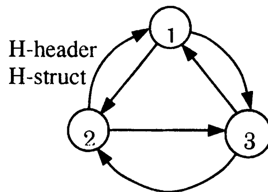

Figure 6. H-header and H-struct exchange

2. After exchange process finished all nodes should have identical global H-header and H-struct, therefore parallel mining in pre-assigned manner as in previous algorithm can be performed.

7. Experimental Result

We have done several experiments to assess the performance of our proposed Hmine-rev algorithm and its parallel version. First we compared performance of H-mine algorithm and our proposed modification, Hmine-rev, and show that Hmine-rev does not suffer any degradation in its performance. Then we did experiments on its parallel version and show that parallel Hmine-rev can significantly increase mining performance.

The data used in all experiments is the following synthetic data generated by Almaden data generator program.

Data: T10I2D100kL2kN1k.
T (avg length of transaction): 10
I (avg length of max. freq. pattern): 2
D (number of transaction): 100,000
L (number of max. freq. pattern): 2,000
N (number of item): 1,000

Resulted H-struct and pattern number is:
H-struct Number: 995,850

L(1): 755	L(4): 42,142	L(7): 258
L(2): 115,162	L(5): 9,061	L(8): 27
L(3): 159,446	L(6): 1,689	L(9): 3

Total patterns: 328,543

7.1. Performance Comparison

Here we did performance comparison between Apriori, FP-Tree, H-mine and Hmine-rev algorithms to mine frequent patterns on artificial sparse data set above. The main objective of this experiment is to compare performance of H-mine and our proposed modification, Hmine-rev. The rest of algorithms are included only as reference. The time needed for each algorithm to complete mining in various minimum support is shown in Figure 7 (time in seconds).

From the result we can see that our proposed Hmine-rev algorithm has comparable performance with original H-mine algorithm in all cases of minimum support. There is no performance loss by removing link adjustment from the algorithm.

7.2. Performance of Parallel Hmine-rev

7.2.1. Pre-assigned Parallelization in Shared-disk Environment

First we did parallel experiment on shared-disk using pre-assigned parallelization technique. However, due to several technical reasons we did not perform experiment in truly shared-disk environment, instead we simulated it using shared-nothing environment with 4 nodes of CPU and only one node is connected to storage device and MPICH is used as parallelization interface. First node that has the access to data set is responsible to build H-header and H-struct and then broadcasting it to all other nodes. Then, after completing mining their assigned queues independently, all other nodes reduces (sends) their results back to first node. Speedup ratio to mine frequent patterns using above data in this way with minimum support 0.5% (500) on different number of nodes is shown in Figure 8.

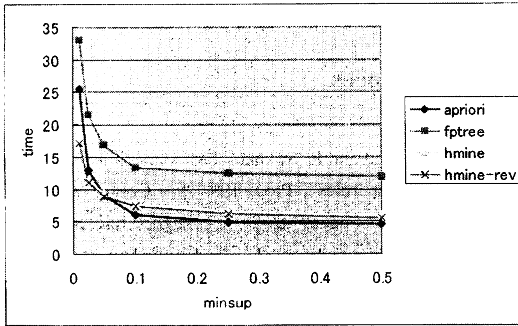

Figure 7. Hmine-rev performance comparison

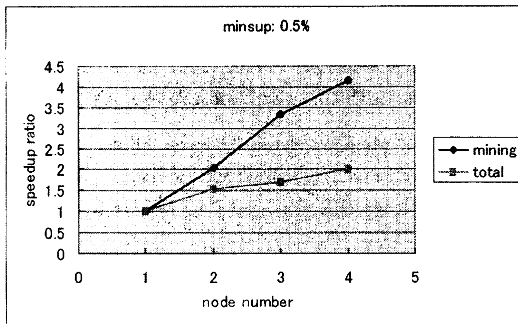

Figure 8. Shared-disk simulated environment

Speedup ratio is the ratio of consumed time between one node and multiple nodes. Mining time is the time needed to mine H-header and H-struct after they are completely constructed in memory. Total time is the sum of H-header and H-struct creation and broadcast time, mining time and results reducing time.

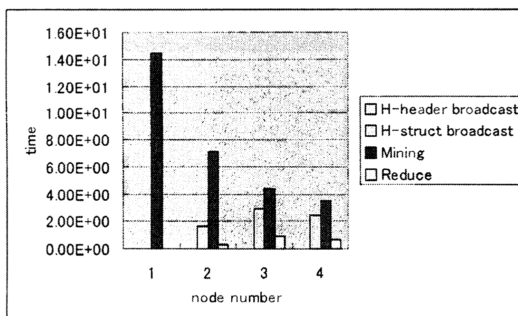

Figure 9. Consumed time comparison

We can see that speedup ratio of our proposed Hmine-rev algorithm is ideal regarding the mining process, i.e. speedup ratio is equal to number of nodes. However, the overall performance is not ideal due to time consumed in broadcasting and reducing process. Consumed time comparison in Figure 9 shows that H-struct broadcasting time become comparable to mining time as node number increases.

Therefore, in truly shared-disk environment where broadcasting and reducing process are not needed, our proposed parallel algorithm can do parallel mining in a very efficient way with speedup ratio near to ideal.

7.2.2. Pre-assigned Parallelization in Shared-nothing Environment

In this experiment the parallel environment is shared-nothing with 8 nodes of CPU and MPICH is used as parallelization interface. We distribute the data above to all nodes such that each node has 12,500 transactions and we use minimum support of 0.01% (10). Each node builds local H-header and H-struct using its own local data, and exchange to each other to construct global ones. The time consumed to mine frequent patterns on above data using non-parallel and parallel algorithm is shown in Table 2 below (time in seconds).

Table 2. Shared-nothing distributed data

	Non-parallel	Parallel (8 CPU)
H-header creation time (& exchange)	2.01668	0.150526
H-struct creation time (& exchange)	5.87495	8.87469
Mining time	43.4081	(max) 6.16498
Total time	51.2997	15.1594

Time difference between the fastest node and the slowest node is about 2 seconds. Mining speedup ratio is about 7 times, and overall speedup ratio is about 3.4 times. Consumed time comparison for each process in Figure 10 shows that for parallel version the time consumed in H-struct creation and exchange process exceeds the time consumed in mining process itself.

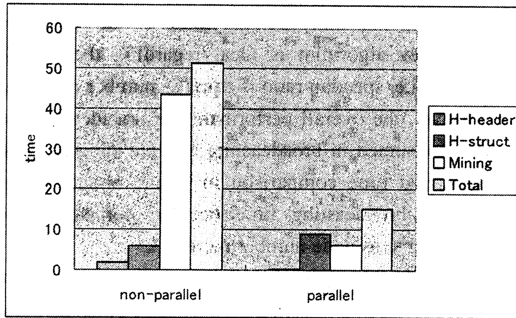

Figure 10. Shared-nothing distributed data

8. Conclusions and Future Works

Hmine-rev that modified link structure and processing order of H-mine algorithm has successfully removed link adjustment from H-mine without any loss in performance and thus make it possible to do parallelization on the algorithm efficiently.

Pre-assigned parallelization technique in a shared-disk environment together with Hmine-rev algorithm that has no link adjustment made it possible to build parallel algorithm that requires no data-exchange nor any communications between each node in the middle of mining process. This allows us to build ideal parallel algorithm in shared-disk environment and further to design parallel algorithm that exchanges data and communicates between each node in an efficient way in shared-nothing environment.

As the future work we plan to compare performance of pre-assigned parallelization technique with a technique that is dynamically assigning unprocessed queue to available idle node in the middle of mining process in round-robin fashion. On the other hand, in shared-nothing distributed-data environment we plan to remove the exchange process of H-header and H-struct between all nodes that took significant time by utilizing some 'external linking' method that connects some parts of data structures between each node. This 'external links' will provide Hmine-rev algorithm with information on where to find the next link of current H-header or H-struct during the mining process.

9. References

[1] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules", Proc. 1994 Int. Conf. Very Large Data Bases , pages 487499, Santiago, Chile, September 1994.

[2] J. Han, J. Pei and Y. Yin, "Mining Frequent Pattern without Candidate Generation", Proc. of the ACM SIGMOD Conf. on Management of Data, 2000.

[3] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang, " H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases ", Proc. 2001 Int. Conf. on Data Mining (ICDM'01)}, San Jose, CA, Nov. 2001.

[4] P. Iko, M. Kitsuregawa, "Parallel FP-growth on PC cluster", Proc. of Seventh Pacific-Asia Conference of Knowledge Discovery and Data Mining (PAKDD03), 2003.

[5] William Gropp and Ewing Lusk, "User's guide for mpich, a portable implementation of MPI", Technical Report ANL-96/6, Argonne National Laboratory, 1994.