Parallel Mining Algorithms for Generalized Association Rules with Classification Hierarchy

Takahiko Shintani
Institute of Industrial Science, The University of Tokyo
shintani@tkl.iis.u-tokyo.ac.jp

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

Abstract

Association rule mining recently attracted strong attention. Usually, the classification hierarchy over the data items is available. Users are interested in generalized association rules that span different levels of the hierarchy, since sometimes more interesting rules can be derived by taking the hierarchy into account.

In this paper, we propose the new parallel algorithms for mining association rules with classification hierarchy on a shared-nothing parallel machine to improve its performance. Our algorithms partition the candidate itemsets over the processors, which exploits the aggregate memory of the system effectively. If the candidate itemsets are partitioned without considering classification hierarchy, both the items and its all the ancestor items have to be transmitted, that causes prohibitively large amount of communications. Our method minimizes interprocessor communication by considering the hierarchy. Moreover, in our algorithm, the available memory space is fully utilized by identifying the frequently occurring candidate itemsets and copying them over all the processors, through which frequent itemsets can be processed locally without any communication. Thus it can effectively reduce the load skew among the processors. Several experiments are done by changing the granule of copying itemsets, from the whole tree, to the small group of the frequent itemsets along the hierarchy. The coarser the grain, the easier the control but it is rather difficult to achieve the sufficient load balance. The finer the grain, the more complicated the control is required but it can balance the load quite well.

We implemented proposed algorithms on IBM SP-2. Performance evaluations show that our algorithms are effective for handling skew and attain sufficient speedup ratio.

1 Introduction

Recently, Data Mining also known as knowledge discovery in databases has attracted strong attention. Because of the progress of data collection tools such as POS, large amount of transaction data have been electronically generated. However such a data has been just archived and has not been used effectively. Data mining is the method of efficient discovery of useful information such as rules and previously unknown patterns existing between data items embedded in large databases, which allows more effective utilization of large amount of accumulated data.

One of the most important problems in data mining is discovery of association rules within a large database. In order to extract association rules, the transaction databases have to be scanned repeatedly. In addition, in order to improve the quality of the rules, we have to handle large amount of raw transaction data instead of samples, which further increases the computation time. In general, it is difficult for a single processor system to provide reasonable response time.

In our previous study, we proposed parallel algorithm for mining association rules on a shared-nothing environment, named HPA(Hash Partitioned Apriori)[SK96]. Association rules are the rules about what items are bought together within a transaction. In order to exploit the parallelism, our algorithm HPA partitions not only the transaction database but also the candidate itemsets among the processors.

So far a few parallel algorithms for association rules are introduced[PCY95, CHN+96, AS96, CNFF96, HKK97]. However, the algorithms such as CD(Count Distribution), PDM (Parallel Data Mining), FDM(Fast Distributed Mining) just partition the transaction database but do not partition the candidate itemsets. Only DD(Data Distribution[HKK97]) method partitions the candidate itemsets but it partition the candidates naively. Large amount of communication is required. After we proposed HPA[SK96], algorithms named IDD(Intelligent Data Distribution) and HD(Hybrid Distribution) are introduced [HKK97], which also partition the candidate itemsets. But these algorithms exchange all the transaction data among the processors. Thus communication overheads are still large. On the other hand, communication is minimized in HPA, since the transaction data is just sent to the processors which are assigned a relevant candidates.

Usually, the classification hierarchy over the data items is available. Users are interested in generating association rules that span different levels of the classification hierarchy, since sometimes more interesting rules can be derived by taking the hierarchy into account which otherwise could not be found out. In [SA95], fundamental algorithm named cumulate was proposed to derive the generalized association rules with classification hierarchy which is a natural extension of non-hierarchical association rule mining algorithm, Apriori[RR94]. The algorithm checks all the combination of

SIGMOD '98 Seattle, WA, USA

items between all the levels of hierarchy. Adding the classification hierarchy further increases the processing complexity, which results in long computation time. Parallel processing is essential to obtain reasonable response time.

In this paper, we propose the parallel algorithms for mining generalized association rules with classification hierarchy on shared-nothing parallel machines. Three different algorithms named NPGM(Non Partitioned Generalized association rule Mining), HPGM(Hash Partitioned Generalized association rule Mining) and H-HPGM(Hierarchical HPGM) are proposed. NPGM does not partition the candidate itemsets. HPGM partitions the candidates using the hash function like HPA but it does not take the hierarchically structural information into account. H-HPGM does utilize the hierarchical structure so that the communication among the processors becomes much reduced. That is, H-HPGM allocates all the items of each individual hierarchy to a single processor. To generalize the rule of bottom level itemsets, the association of the ancestors of individual itemset are examined. In H-HPGM, the inter-processor communication can be eliminated, since the unit of allocation is each hierarchy. In addition, we also propose another three different algorithms to exploit the parallelism more efficiently. Although the H-HPGM can reduce the communication overhead significantly, the granule of the candidate allocation is too coarse to balance the load among the processors. In order to obtain sufficiently high speed up ratio, we have to make the loads of each processor as even as possible. Load skew is intrinsic to the data mining problem, since rules are derived due to the difference of occurring frequency of itemsets. We believe load balancing problem in parallel data mining is one of the most important and challenging issues. In the case the aggregate size of the candidate itemsets is smaller than the total capacity of the system memory, there remains free space. We could utilize this unused space to improve load balancing. We duplicate the frequently occurring itemsets to all the processors, through which communication caused by frequent itemset counting can be performed locally and the communication can be further reduced. In this paper, we designed three different algorithm by changing the size of duplication granule, H-HPGM-TGD(H-HPGM with Tree Grain Duplicate), H-HPGM-PGD(H-HPGM with Path Grain Duplicate) and H-HPGM-FGD(H-HPGM with Fine Grain Duplicate). H-HPGM-TGD duplicates the candidate itemsets in the unit of hierarchy. When the size of free memory is small, H-HPGM-TGD cannot duplicate the candidate itemsets, since it copies the whole hierarchy and its size is really large. H-HPGM-PGD selects the frequently occurring leaf itemsets and duplicates them and their all ancestor itemsets. The duplication granule in H-HPGM-PGD is smaller compared with H-HPGM-TGD. Load can be better balanced. The final algorithm, H-HPGM-FGD, detects the frequently occurring candidate itemsets which consists of itemsets of any level. Although the granule of this algorithm is finest among three proposed algorithms and potentially it can balance the load in the finest way, the algorithm becomes more complicated. The coarser the granule, the simpler and the easier the algorithm to implement.

We implemented these algorithms on a shared-nothing parallel machine to examine their tradeoff. Detail performance evaluation results are given and it shows that our final algorithm can exploits the parallelism most effectively and is very effective for handling skew.

This paper is organized as follows. In next section, we explain the problem of mining generalized association rules with classification hierarchy. In section 3, we propose our parallel algorithms. Performance evaluations is given in section 4. Section 5 concludes the paper.

2 Mining Generalized Association Rules with Classification Hierarchy

First we introduce some basic concepts of generalized association rules, using the formalism presented in [SA95]. Let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of literals, called items. Let $\mathcal{T}$ be a classification hierarchy on the items, which organize relationships of items in a tree form, shown in Figure 1.

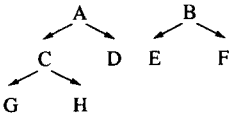

Figure 1: The classification hierarchy

An edge in $\mathcal{T}$ represents an *is-a* relationship. If there is an edge in $\mathcal{T}$ from x to y, we call x a parent of y and y a child of x. If there is an edge from x to z in a transitive-closure, we call x an ancestor of z and z a descendant of x. In Figure 1, A is a parent of C, C is a child of A, C is a parent of G and G is a child of C. Further A is an ancestor of G and G is a descendant of A. Since the classification hierarchy is acyclic, there is no item which is an ancestor of itself. Let $\mathcal{D} = \{t_1, t_2, \ldots, t_n\}(t_i \subseteq \mathcal{I})$ be a set of transactions, where each transaction t has an associated unique identifier called TID. We say a transaction t *contains* a set of items X, if X is in t or is an ancestor of some item in t. The itemset X has *support* s in the transaction set $\mathcal{D}$, if s% of transactions in $\mathcal{D}$ contain X, here we denotes $s = sup(X)$. An *generalized association rules with classification hierarchy* is an implication of the form $X \Rightarrow Y$, where $X, Y \subset \mathcal{I}$, $X \cap Y = \phi$ and no item in Y is an ancestor of any item in X. Each rule has two measures of value, *support* and *confidence*. The *support* of the rule $X \Rightarrow Y$ is $sup(X \cup Y)$. The *confidence* c of the rule $X \Rightarrow Y$ in the transaction set $\mathcal{D}$ means c% of transactions in $\mathcal{D}$ that contain X also contain Y, which can be written as the ratio $sup(X \cup Y)/sup(X)$. Here a rule $x \Rightarrow ancestor(x)$ is redundant, since its confidence is always 100%.

The problem of mining generalized association rules with classification hierarchy is to find all the rules that satisfy a user-specified minimum support(min_sup) and minimum confidence(min_conf) on the assumption that we are given a set of transactions $\mathcal{D}$ and a classification hierarchy over the items. This problem can be decomposed into two subproblems:

1. Find all itemsets that have support above the user-specified minimum support. These itemsets are called the *large itemsets* and the other itemsets are called *small itemsets*. The items which contained large itemset are called *large items* and the other items are called *small items*.

2. For each large itemset, derive all rules that have more than user-specified minimum confidence as follows: for large itemset X and any $Y (Y \subset X)$, if $support(X)/support(X - Y) \geq min\_conf$, then the rule $(X - Y) \Rightarrow Y$ is derived.

After finding all large itemsets, the association rules are derived in a straightforward manner. This second subproblem is not a big issue. However because of the large scale

of transaction data sets used in data mining, the first subproblem is a nontrivial problem.

Here we explain the Cumulate algorithm for finding all large itemsets, proposed in [SA95]. In the first pass (pass 1), sup_cou for each item is counted by scanning the transaction database. Hereafter we prepare a field named sup_cou for each itemset, which is used to measure how many times the itemset contained in transactions. Since itemset here contains just single item, each item has a sup_cou field. All the items which satisfy the minimum support are picked out. These items are called large item (L_1). Hereafter k-itemset is defines a set of k items. The second pass (pass 2), the 2-itemsets are generated using L_1 which is called the candidate 2-itemsets (C_2), and delete any candidate in C_2 that consists of an item and its ancestor. Note that we need not count any itemset which contains both an item and its ancestor. Delete any ancestors in $\mathcal{T}$ that are not present in any of the candidates in C_2. Note that we can drop ancestors that are not present in any of the candidates at the same time. Then the sup_cou of C_2 is counted by scanning the transaction database. Here sup_cou of the itemset means the number of transactions which contain the itemset. At the end of scanning the transaction data, the large 2-itemsets (L_2) which satisfy minimum support are determined. The following denotes the k-th iteration, pass k ($k \geq 2$).

1. Generate candidate itemsets:
   The candidate k-itemsets (C_k) are generated using large $(k-1)$-itemsets (L_{k-1}) as follows: join L_{k-1} with L_{k-1} and delete all the k-itemsets whose some of the $(k-1)$-itemsets are not in L_{k-1}. If k is 2, delete any candidates in C_2 that consists of an item and its ancestor. Delete any ancestors in $\mathcal{T}$ that are not present in any of the candidates in C_k.

2. Count support:
   Read the transaction database, add all ancestors of the items in a transaction t that are present in $\mathcal{T}$ to t. Increment the sup_cou of all candidates in C_k that are contained in t.

3. Determine large itemsets:
   The candidate k-itemsets are checked for whether they satisfy the minimum support or not, the large k-itemsets (L_k) which satisfy the minimum support are determined.

This procedure terminates when the large itemset becomes empty.

3 Parallel Algorithms

In this section, we describe parallel algorithms for the first subproblem defined in the previous section, which we call count support processing hereafter. In the sequential algorithm, the count support processing requires the longest computation time, where the transaction database is scanned and a large number of candidate itemsets are examined.

If the size of all the candidate itemsets is smaller than the size of the memory of each processor, all the processor can hold whole candidate itemsets. In such a case, parallelization is straightforward. By partitioning the transaction database over all the nodes, the transaction data can be read and candidate itemsets can be counted in parallel. However for large scale transaction data sets, this assumption does not hold. In mining generalized association rules, the associations between all the possible ancestors of items have to be examined. Thus the amount of candidate itemsets becomes considerably larger compared with usual non-hierarchical association rule mining. In the case where the candidate itemsets do not fit in the local memory of a single node, the candidate itemsets are partitioned into fragments, each of which fits in the memory size of a node. The transaction database has to be scanned for each fragment. Thus such repetitive scanning of transaction database incurs the excessive I/O's and degrades the performance significantly. Our algorithms partition the candidate itemsets over the memory space of all the nodes to exploit the aggregate memory of the system. For simplicity, we assume that the size of the candidate itemsets is larger than the size of local memory of single node but is smaller than the sum of the memory space of all the nodes. It is easy to expand this algorithm to handle the candidate itemsets whose size exceeds the sum of all the nodes memories.

3.1 Non Partitioned Generalized association rule Mining : NPGM

In NPGM, the candidate itemsets are copied over all the nodes, each node can work independently and the final statistics are gathered into a coordinator node where minimum support conditions are examined. Figure 2 gives the behavior of pass k of the n-th node, using the notation in Table 1.

$\mathcal{L}_k$	Set of all the large k-itemsets.
$\mathcal{C}_k$	Set of all the candidate k-itemsets.
$\|\mathcal{C}_k\|$	The size of $\mathcal{C}_k$ in bytes.
M	The size of main memory in bytes.
$\mathcal{D}^n$	Transactions stored in the local disk of the n-th node.
C_k^d	Sets of fragment of candidate k-itemsets. Each fragment fits in the local memory of a node. $(d = 1, \ldots, \lceil \|\mathcal{C}_k\|/M \rceil,\ \mathcal{C}_k = \bigcup_{d=1}^{\lceil \|\mathcal{C}_k\|/M \rceil} C_k^d)$
$\|C_k^d\|$	The size of C_k^d in bytes.
L_k^d	Sets of large k-itemsets derived from C_k^d.

Table 1: Notation

Each node works as follows:

1. Generate the candidate itemsets:
   Each node generates the candidate k-itemsets ($\mathcal{C}_k$) using the large $(k-1)$-itemsets ($\mathcal{L}_{k-1}$). If k is 2, delete the candidates that contains an items and its ancestor. Insert $\mathcal{C}_k$ into the hash table, and delete any ancestors in $\mathcal{T}$ that are not present in any of the candidates in $\mathcal{C}_k$.

2. Scan the transaction database and count the sup_cou value:
   Each node reads the transaction database from its local disk, generates extended transaction t' by adding all ancestors of the items in a transaction t that are present in $\mathcal{T}$ to t'.
   Increment the sup_cou of all candidates in C_k that are contained in t'.

3. Determine the large itemsets:
   After reading all the transaction data, all node's sup_cou are gathered into the coordinator node and checked to determine whether the minimum support condition is satisfied or not.

$k \geq 2$
$\mathcal{C}_k :=$ The candidates of size k generated from $\mathcal{L}_{k-1}$.
if $(k = 2)$ **then**
  Delete the candidates that contains an items and its ancestor.
Delete any ancestors in $\mathcal{T}$ that are not present in $\mathcal{C}_k$.
$\{C_k^d\} :=$ Partition $\mathcal{C}_k$ into fragments each of in a node's local memory$(d=1, \ldots, \lceil |\mathcal{C}_k|/M \rceil)$.
for $(d=1;\ d \leq \lceil |\mathcal{C}_k|/M \rceil;\ d{+}{+})$ **do**
  forall $t \in \mathcal{D}^n$ **do**
    $t\prime :=$ Add all ancestors of item $x(\in t)$ that are present in the candidates in $\mathcal{C}_k$.
    Increment the sup_cou of all candidates in C_k^d that are contained in $t\prime$.
  end
  Send the sup_cou of C_k^d for to the coordinator node.
  /* Coordinator node determine L_k^d which satisfy */
  /* user-specified minimum support in C_k^d and broad- */
  /* cast L_1^d to all nodes. */
  Receive L_k^d from the coordinator node.
end
$\mathcal{L}_k := \bigcup_d L_k^d$

Figure 2: NPGM algorithm

4. If large k-itemset is empty, the algorithm terminates. Otherwise the coordinator node broadcasts large k-itemsets to all the nodes, $k := k + 1$ and goto "1".

This algorithm is very simple and easy to implement, where no transaction data have to be exchanged among the nodes in the count_support phase. However, if the size of all the candidate itemsets exceeds the local memory of a single node, the candidate itemsets are partitioned into fragments, each of which can fits within the local memory of a single node, and the above process is repeated for each fragment. In the Figure 2, the first level for-loop shows this. The disk I/O becomes prohibitively costly when the candidate itemsets becomes large.

3.2 Hash Partitioned Generalized association rule Mining : HPGM

In NPGM, the candidate itemsets are not partitioned but just copied among the nodes. However the candidate itemsets usually becomes too large to fit within the local memory of a single node. If it is partitioned naively, transaction data has to be broadcast to all the other nodes. HPGM partitions the candidate itemsets among the nodes using a hash function like in the hash join, which eliminate broadcasting. Figure 3 gives the behavior of pass k by the n-th node, using the notation in Table 2.

$\mathcal{L}_k$	Set of all the large k-itemsets.
$\mathcal{C}_k$	Set of all the candidate k-itemsets.
$\mathcal{D}^n$	Transactions stored in the local disk of the n-th node.
C_k^n	Sets of candidate k-itemsets whose hashed value is corresponding to n-th node. ($\mathcal{C}_k = \bigcup_{p=1}^{N} C_k^n$, N means the number of nodes)
L_k^n	Sets of large k-itemsets derived from C_k^n.

Table 2: Notation

$k \geq 2$
$\mathcal{C}_k :=$The candidates of size k generated from L_{k-1}.
if $(k = 2)$ **then**
  Delete the candidates that contains an items and its ancestor.
Delete any ancestors in $\mathcal{T}$ thar are not present in $\mathcal{C}_k$.
$\{C_k^n\} :=$ All the candidate k-itemsets, whose hashed value corresponding to the n-th node.
forall $t \in \mathcal{D}^n$ **do**
  $t' :=$ Add all ancestors of $x(\in t)$ that are present in the candidates in $\mathcal{C}_k$.
  forall k-itemset $x \in t\prime$ **do**
    Determine the destination node ID by applying the same hash function which is used in item partitioning, and send that k-itemset to it. If it is its own id, increment the sup_cou for the itemset.
    Receive k-itemset from the other nodes and increment the sup_cou for that itemset.
  end
end
$\{L_k^n\} :=$All the candidates in C_k^n with minimum support.
Send L_k^n to the coordinator node.
/* Coordinator node make up $\mathcal{L}_k := \bigcup_n L_k^n$ and broad- */
/* cast to all the nodes. */
Receive $\mathcal{L}_k$ from the coordinator node.

Figure 3: HPGM algorithm

Each node works as follows:

1. Generate candidate itemsets:
   Each node generates the candidate k-itemsets ($\mathcal{C}_k$) using the large $(k-1)$-itemsets ($\mathcal{L}_{k-1}$). If k is 2, delete the candidates that contains an items and its ancestor. Delete any ancestors in $\mathcal{T}$ that are not present in any of the candidates in $\mathcal{C}_k$.
   Apply the hash function to the candidates in $\mathcal{C}_k$ and determine the destination node ID. If the ID is its own, insert it into the hash table.

2. Scan the transaction database and count the sup_cou value:
   Each node reads the transaction database from its local disk and generates extended transaction t' by adding all ancestors of the items in a transaction t that are present in $\mathcal{T}$.
   Generate k-itemsets from t' and apply the same hash function used in phase 1. Derive the destination node ID and send the k-itemset to it. For the itemsets received from other nodes and those locally generated whose ID equals the node's own ID, search the hash table. If hit, increment its sup_cou value.

3. Determine the large itemsets:
   After reading all the transaction data, each node can determine individually whether each candidate in C_k^n satisfy minimum support or not. Each node send L_k^n to the coordinator node, where all the large k-itemsets $\mathcal{L}_k := \bigcup_n L_k^n$ are derived.

4. If large k-itemset is empty, the algorithm terminates. Otherwise the coordinator node broadcasts large k-itemsets to all the nodes, $k := k + 1$ and goto "1".

In this algorithm, the candidate itemsets are partitioned among the nodes. However, classification hierarchy is not taken into account at all. Each node has to send all the ancestor itemsets in addition to the leaf level itemsets. The

destination node for each item is randomly determined just by hashing. This incurs considerable communication cost. Figure 4 shows an example of HPGM.

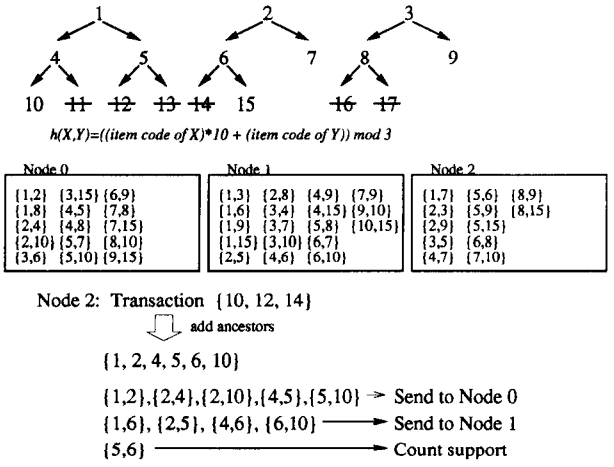

Figure 4: Example of HPGM

Example 1: Assuming there are three nodes in a system. Let the large items, derived at pass 1, be $\mathcal{L}_1 = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15\}$, which organize the classification hierarchy in Figure 4. Let the hash function be $hash(X, Y) = ((item\ code\ of\ X)*10+(item\ code\ of\ Y))\ mod\ 3\ where\ (item\ code\ of\ X) \leq (item\ code\ of\ Y)$. Therefor, the candidate itemsets are partitioned as shown in Figure 4. Suppose Node 2 reads the transaction $t = \{10, 12, 14\}$. Then, Node 2 adds all the ancestors in t and generate new transaction $t' = \{1, 2, 4, 5, 6, 10\}$. Node 2 generates 2-itemsets from t' and applies the same hash function used at the candidate partitioning. For example, since $\{1, 2\}$ is generated and its derived hashed value is 0, Node 2 sends $\{1, 2\}$ to Node 0. For this transaction, Node 2 sends 18 items to the other nodes. Thus a lot of communication is required.

3.3 Hierarchical HPGM : H-HPGM

H-HPGM partitions the candidate itemsets among the nodes taking the classification hierarchy into account so that all the candidate itemsets whose root items are identical be allocated to the identical node, which eliminates communication of the ancestor items. Thus the communication overhead can be reduced significantly compared with original HPGM.

Figure 5 gives the behavior of pass k by n-th node, using the notation in Table 3.

$\mathcal{L}_k$	Set of all the large k-itemsets.
$\mathcal{C}_k$	Set of all the candidate k-itemsets.
$\mathcal{D}^n$	Transactions stored in the local disk of the n-th node.
C_k^n	Sets of candidate k-itemsets whose hashed value calculated with their root item is corresponding to n-th node.
L_k^n	Sets of large k-itemsets derived from C_k^n.

Table 3: Notation

Each node works as follows:

1. Generate candidate itemsets:
   Each node generates $\mathcal{C}_k$ using $\mathcal{L}_{k-1}$.
   (1) $k \geq 2$
   (2) $\mathcal{C}_k$:=The candidates of size k generated from L_{k-1}.
   (3) **if** $(k = 2)$ **then**
   (4) Delete the candidates that contains an item and its ancestor.
   (5) Delete any ancestors in $\mathcal{T}$ that are not present in $\mathcal{C}_k$.
   (6) $\{C_k^n\}$:= All the candidate k-itemsets, whose hashed value calculated with its root items corresponding to the n-th node.
   (7) **forall** $t \in \mathcal{D}^n$ **do**
   (8) t':= Replace the item in t with the large item in its ancestors which is closest to the bottom if there are small items.
   (9) **foreach** n-th node **do**
   (10) t'' := Select all items whose root items are allocated to n-th node.
   (11) **if** (n = own node ID) **then**
   (12) Generate k-itemset from t'', and increment the sup_cou for the itemset and all its ancestor candidates.
   (13) **else**
   (14) Send t'' to n-th node.
   (15) Receive items from the other nodes.
   (16) Generate k-itemset from receive items, and increment the sup_cou for the itemset and all its ancestor candidates .
   (17) **end**
   (18) **end**
   (19) $\{L_k^n\}$:= All the candidates in C_k^n with minimum support.
   (20) Send L_k^n to the coordinator node.
   (21) /* Coordinator node make up $\mathcal{L}_k := \bigcup_p L_k^n$ and */
   /* broadcast to all the nodes. */
   (22) Receive $\mathcal{L}_k$ from the coordinator node.

Figure 5: H-HPGM algorithm

   If k is 2, delete the candidates that contains an items and its ancestor.
   Delete any ancestors in $\mathcal{T}$ that are not present in any of the candidates in $\mathcal{C}_k$.
   Apply the hash function to the candidate itemsets in $\mathcal{C}_k$. Here each item of the candidate itemset is replaced by its root items, then hash function is applied and destination node ID is determined. If the ID is its own, insert it into the hash table.

2. Scan the transaction database and count the sup_cou value:
   Each node reads the transaction database from its local disk and generates extended transaction t' by replacing the item in t with the large item in its ancestors which is closest to the bottom, if there are small items. For each node n, select all items in t' whose root item is allocated to n-th node and send them to n-th node. For the itemsets received from other nodes and those locally generated whose root item is allocated to own, generate k-itemset from the itemsets and increment the sup_cou value of this k-itemset and its all ancestor candidates.

3. – 4. Same as in HPGM.

In Figure 6, Example 2 illustrates the reduction of communications in H-HPGM under the same condition at Example 1.

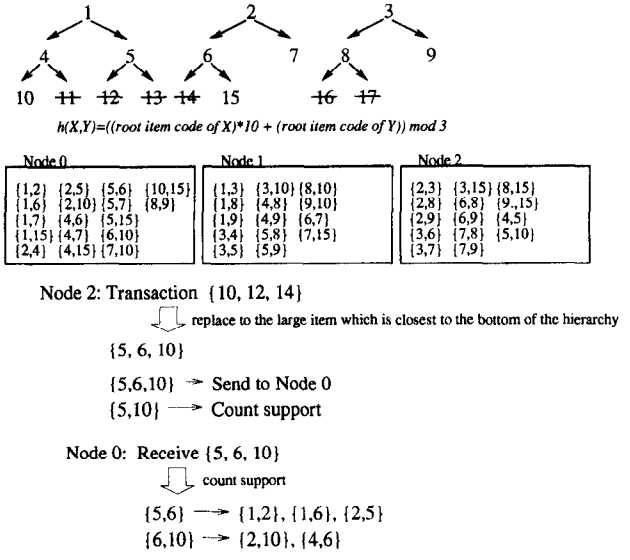

Figure 6: Example of H-HPGM

Example 2: Let the hash function be $hash(X,Y) = ((root\ item\ code\ of\ X) * 10 + (root\ item\ code\ of\ Y))\ mod\ 3$ *where* $(root\ item\ code\ of\ X) \leq (root\ item\ code\ of\ Y)$. Therefore, the candidate itemsets whose root candidate is $\{1,2\}$ are allocated to Node 0. The other candidate itemsets are allocated in the same way. As a result, the candidate itemsets are partitioned as shown in Figure 6. Suppose Node 2 reads the transaction $t = \{10,12,14\}$. Then, Node 2 generates extended transaction $\{5,6,10\}$, sends the itemset $\{5,6,10\}$ to Node 0, and increments the sup_cou of $\{6,10\}$ and its all ancestor candidates $\{2,10\},\{4,6\}$. Then Node 2 generates extended transaction $\{5,6,10\}$, increments the sup_cou value of $\{6,10\}$ and its all ancestor candidates $\{2,10\}$, $\{4,6\}$. Since the root item of $\{5\}$ and $\{10\}$ are $\{1\}$ and $\{6\}$ is $\{2\}$ and the root candidate $\{1,2\}$ is allocated to Node 0, Node 2 sends the itemset $\{5,6,10\}$ to Node 0. Node 0 receives the itemsets $\{5,6,10\}$ from Node 2, generates 2-itemsets $\{5,6\}$, $\{6,10\}$, and increments the sup_cou of $\{5,6\}$, $\{6,10\}$ and their all ancestor candidates.

In H-HPGM, Node 2 sends 3 items to the other nodes for this transaction. On the other hand, Node 2 sends 18 items for the same transaction in HPGM. This shows that it is very effective to take the classification hierarchy information into account on the candidate itemset partitioning for reducing the communication overhead.

3.4 Skew Handling

In the case the size of the candidate itemsets is smaller than the available system memory, H-HPGM does not use the remaining free space. Since H-HPGM partitions the candidate itemsets in the unit of hierarchy of the candidate itemsets, the grain is too coarse to achieve a flat workload distribution. If the transaction data is skewed, that is, there are some itemsets which appear very frequently in the transaction data, the node which is allocated such itemsets will receive a lot of transaction data, which incurs a system bottleneck.

In this section, we present three methods to handle this problem by identifying such frequent itemsets and treating them in an appropriate manner.

3.4.1 H-HPGM with Tree Grain Duplicate : H-HPGM-TGD

H-HPGM partitions the candidate itemsets among the nodes so that all the candidate itemsets whose root items are the same be allocated to the identical node. That is, H-HPGM divides the candidate itemsets into the hierarchy of the candidate itemsets and allocates such whole hierarchy to a node. Thus the granule is a hierarchy, that is, a tree. H-HPGM-TGD detects the tree whose candidate itemsets contain very frequently occurred items, duplicates them among the nodes and counts the sup_cou locally for those itemsets like in NPGM. The behavior of pass k of the n-th node is obtained by replacing the lines (6), (8) and (21) in Figure 5 with Figure 7, using the notation in Table 3 and 4.

C_k^n	Sets of candidate k-itemsets whose hashed value calculated with their root item is corresponding to n-th node.
L_k^n	Sets of large k-itemsets derived from C_k^n.
C_k^D	Sets of candidate k-itemsets which are copied all the nodes.
L_k^D	Sets of large k-itemsets derived from C_k^D.

Table 4: Notation

(6.0) Count the number of descendant candidate for each root k-itemset and the number of candidates allocated for each node by generating the k-itemsets using $\mathcal{L}_k$.
(6.1) Generate k-itemsets from root items.
(6.2) Sort the root itemsets based on their frequency of appearance.
(6.3) $\{C_k^D\}$:=All the root k-itemsets whose frequency is high so as to use the memory space fully.
(6.4) $\{C_k^D\}$:=All the candidate k-itemsets whose root itemset is contained in C_k^D.
(6.5) Delete the candidates in C_k^D from $\mathcal{C}_k$.
(6.6) $\{C_k^n\}$:= All the candidate k-itemsets in $\mathcal{C}_k$, whose hashed value calculated with its root items corresponding to the n–th node.

(8.0) t':=Replace the items in t with the large item in its ancestors which is closest to the bottom, if there are small items.
(8.1) Increment the sup_cou of all candidates in C_k^D that are contained in t'.

(21.0) Send the sup_cou of C_k^D to the coordinator.
(21.1) /* Coordinator determine L_k^D which satisfy user- */
/* specified minimum support in C_k^D. */
(21.2) /* Coordinator node make up $\mathcal{L}_k := L_k^D + \bigcup_p L_k^p$ */
/* and broadcast to all the nodes. */

Figure 7: Algorithm H-HPGM-TGD

Each node works as follows:

1. Generate the candidate itemsets:
   (a) Count the number of descendant candidates for each root k-itemset and the number of candidates

allocated for each node by generating the k-itemsets using L_{k-1}.

(b) Generate k-itemsets from root items. Here, these k-itemsets contain the itemsets consisting of the same items, such as $\{1,1\}$.

(c) Sort the root itemsets based on whose item's frequency of appearance.

(d) Choose the most frequently occurring root itemsets and insert them and their descendant candidate itemsets to C_k^D so that all the memory space is used.

(e) Apply the hash function to the remaining candidate itemsets and determine the destination node ID. If the ID is its own, insert it into the hash table (C_k^n).

2. Scan the transaction database and count the sup_cou value:
Each node reads the transaction database from its local disk and generates extended transaction t' by replacing the item in t with the large item in its ancestors which is closest to the bottom, if there are small items. For each node n, select all items in t' whose root item is allocated to n-th node and send them to the n-th node. For the itemsets received from the other nodes and locally generated items whose root item is assigned to its own node to own, generate k-itemset from the itemsets and increment sup_cou value for this k-itemset and its all ancestors.

3. Determine the large itemsets:
After reading all the transaction data, each node can determine individually whether each candidate in C_k^n satisfy minimum support or not. Each node send L_k^n to the coordinator node. The sup_cou of C_k^D of all the nodes are gathered into the coordinator node. It is checked whether the minimum support condition is satisfied or not. The coordinator node determines the large k-itemset(L_k^D) in C_k^D and derives all the large k-itemsets $\mathcal{L}_k = L_k^D + \bigcup_n L_k^n$.

4. Same as NPGM.

Figure 8 shows an example of H-HPGM-TGD.

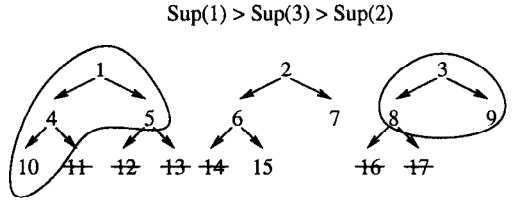

Figure 8: Example of H-HPGM-TGD

Example 3: Assume the large items at pass 1, be $\mathcal{L}_1 = \{1,2,3,4,5,6,7,8,9,10,15\}$ which form the classification hierarchy as shown in Figure 8. H-HPGM-TGD picks up the root items and sorts the root items over the support value of each item. Let the support of each root item be $Sup(1) > Sup(3) > Sup(2)$. H-HPGM-TGD duplicates the candidate itemsets $\{4,5\}, \{5,10\}$ which are included in the tree of root item 1. When there still remains free space, the candidate itemsets which are generated by joining the items in the tree rooted by 1 and in the tree rooted by 3.

3.4.2 H-HPGM with Path Grain Duplicate : H-HPGM-PGD

H-HPGM-TGD duplicates the candidate itemsets in the unit of whole hierarchy. The granule is too coarse to obtain sufficient load balance when the size of free space is small. H-HPGM-PGD picks up the leaf large items and sorts them based on their support value. Then it chooses the most frequently occurring itemsets and copies them and their all ancestor itemsets over all the nodes. Since the granule employed in H-HPGM-PGD is smaller than that of H-HPGM-TGD, it can balance the load among the nodes more effecting. The behavior of pass k of the n-th node is obtained by replacing the lines (6), (8) and (21) of Figure 5. with Figure 9, using the notation in Table 3 and 4.

(6.0) Count the number of candidates allocated for each node.
(6.1) Sort the lowest large items based on their frequency of appearance.
(6.2) Generate k-itemsets from the lowest items.
(6.3) $\{C_k^D\}$:=All the lowest candidate k-itemsets whose frequency is high and their ancestor candidates so as to use the memory space fully.
(6.4) Delete the candidates in C_k^D from $\mathcal{C}_k$.
(6.5) $\{C_k^n\}$:= All the candidate k–itemsets, whose hashed value calculated with its root items corresponding to the n–th node.

(8.0) t':=Replace the items in t with the large item in its ancestors which is closest to the bottom, if there are small items.
(8.1) Increment the sup_cou of all candidates in C_k^D that are contained in t.

(21.0) Send the sup_cou of C_k^D to the coordinator.
(21.1) /* Coordinator determine L_k^D which satisfy user-*/
/* specified minimum support in C_k^D */
(21.2) /* Coordinator node make up $\mathcal{L}_k := L_k^D + \bigcup_p L_k^p$*/
/* and broadcast to all the nodes. */

Figure 9: Algorithm H-HPGM-PGD

Each node works as follows:

1. Generate the candidate itemsets:

(a) Count up the number of candidates allocated for each node by generating the k-itemsets using $\mathcal{L}_{k-1}$.

(b) Pick up the large items in $\mathcal{L}_{k-1}$ which is the closest to the bottom, and sort them based on their support value.

(c) Choose the first several most frequently occurring items using the sorted list derived at (b), and insert it and its all ancestor candidates to C_k^D so that the free memory space is occupied as much as possible.

(d) Delete the candidates in C_k^D from $\mathcal{C}_k$. Insert the candidates in $\mathcal{C}_k$ if its root itemset's hashed value is corresponding to own node ID.

2. Scan the transaction database and count the sup_cou value:
Each node reads the transaction database from its local disk and generates extended transaction t' by replacing items in t with the large item in its ancestors

which is closest to the bottom, if there are small items in t. Increment the sup_cou of all candidates in C_k^D that are contained in t'. For each node n, select all items in t' whose root item is allocated to the n-th node and send them to the n-th node. For the itemsets received from the other nodes and locally generated items whose root is allocated to own node, generate k-itemset from the itemsets and increment sup_cou of this k-itemset and its all ancestors.

3. – 4.
Same as H-HPGM-TGD

Figure 10 shows an example of H-HPGM-PGD for the same condition of Example 3.

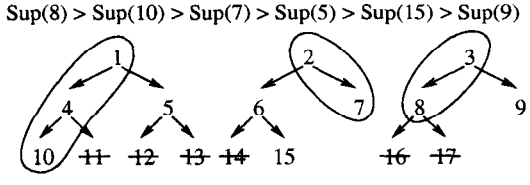

Figure 10: Example of H-HPGM-PGD

Example 4: H-HPGM-PGD picks up the leaf items and sorts the leaf items based on the sup_cou value. Let the support of leaf items be $Sup(8) > Sup(10) > Sup(7) > Sup(5) > Sup(15) > Sup(9)$. H-HPGM-PGD duplicates the candidate itemsets $\{8, 10\}$ and its all ancestor candidate itemsets $\{1,3\}, \{1,8\}, \{3,4\}, \{3,10\}, \{4,8\}$. If there still remains free space, it further duplicates the candidate itemset $\{7,8\}$ and its all ancestors.

3.4.3 H-HPGM with Fine Grain Duplicate : H-HPGM-FGD

H-HPGM-PGD duplicates the candidate itemsets with the smaller grain than H-HPGM-TGD. H-HPGM-PGD examines the candidate itemsets based on the frequency of the leaf large item. It cannot attain good load distribution in the case that there are some internal items whose frequency is high but whose descendant item's frequency is low.

H-HPGM-FGD checks the frequently occurring itemsets which consists of the any level items. It duplicates them and their all ancestor itemsets over all the nodes. Thus only the frequent itemsets are duplicated. The granule becomes finer. But the algorithm is further complicated.

The behavior of pass k by n-th node is obtained by replacing the line (6), (8) and (21) of Figure 5 with Figure 11, using the notation in Table 3 and 4.

Each node works as follows:

1. Generate the candidate itemsets:
   (a) Count up the number of candidates allocated to each node by generating the k-itemsets using $\mathcal{L}_{k-1}$.
   (b) Sort the large items based on their count support value.
   (c) Choose the first most frequently occurring candidate itemsets, and insert them and their all ancestor candidates to C_k^D so that free space be occupied as much as possible.
   (d) Delete the candidates in C_k^D from $\mathcal{C}_k$. Insert the candidates in $\mathcal{C}_k$ if its root itemset's hashed value is corresponding to its own node ID.

(6.0) Count the number of candidates allocated for each node.
(6.1) Sort the large items based on their frequency of appearance.
(6.2) Generate k-itemsets from the large items. in order of the frequency of appearance.
(6.3) $\{C_k^D\}$:=All the candidate k-itemsets whose frequency is high and their ancestor candidates so as to use the memory space fully.
(6.4) Delete the candidates in C_k^D from $\mathcal{C}_k$.
(6.5) $\{C_k^n\}$:= All the candidate k-itemsets, whose hashed value calculated with its root items corresponding to the n–th node

(8.0) t':=Replace the items in t with the large item in its ancestors which is closest to the bottom, if there are small items.
(8.1) Increment the sup_cou of all candidates in C_k^D that are contained in t.

(21.0) Send the sup_cou of C_k^D to the coordinator.
(21.1) /* Coordinator determine L_k^D which satisfy user-*/
/* specified minimum support in C_k^D. */
(21.2) /* Coordinator node make up $\mathcal{L}_k := L_k^D + \bigcup_p L_k^p$*/
/* and broadcast to all the nodes. */

Figure 11: H-HPGM-FGD algorithm

2. Scan the transaction database and count the sup_cou value:
Each node reads the transaction database from its local disk and generates extended transaction t' by replacing items in t with the large item in its ancestors which is closest to the bottom if there are small items in t. Increment the sup_cou of all candidates in C_k^D that are contained in t'. For each node n, select all items in t' whose root item is allocated to the n-th node and send them to the n-th node. For the itemsets received from other nodes and locally generated itemsets whose root is allocated to its own node, generate k-itemset from the itemsets and increment sup_cou of this k-itemset and its all ancestors.

3. – 4.
Same as H-HPGM-TGD

Figure 12 shows an example of H-HPGM-FGD under the same condition of Example 3.

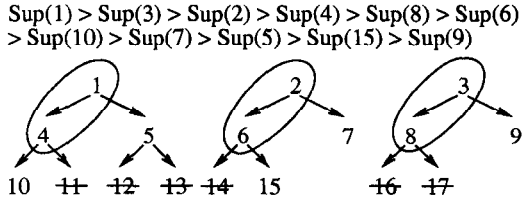

Figure 12: Example of H-HPGM-FGD

Example 5: Let the support of large items be $Sup(1) > Sup(3) > Sup(2) > Sup(4) > Sup(8) > Sup(6) > Sup(10) > Sup(7) > Sup(5) > Sup(15) > Sup(9)$. H-HPGM-FGD duplicates the candidate itemsets according to decreasing order of the support count value so that remaining free space can be occupied as much as possible. In this example, the

candidate itemsets $\{4,8\}, \{4,6\}, \{6,8\}$ and their all ancestor candidate itemsets are duplicated.

4 Performance Evaluation

We implemented all the above algorithms on IBM 16-node SP-2 and measured the performance of each method. SP-2 employs a shared-nothing parallel architecture. Each node contains a POWER2 processor, 256MB local memory and a 2GB local disk drive. HPS(High-Performance Switch) inter connects the nodes together.

To evaluate the performance of the proposed parallel algorithms, synthetic datasets emulating retail transactions are used. The generation procedure is based on the method described in [SA95]. Table 5 shows the meaning of the various parameters and the characteristics of the dataset used in our experiments.

4.1 Comparison of HPGM and H-HPGM

First, we show the performance comparison between HPGM and H-HPGM. Figure 13 shows the execution time of HPGM and H-HPGM of pass 2 varying the value of minimum support.

Table 6 shows the average amount of received messages of HPGM and H-HPGM at pass 2 in each node, where the synthetic data R30F5 was used with 0.3% minimum support. As you can see from the Table, the amount of communications of H-HPGM is much smaller than that of HPGM.

# of nodes	Average amount of received messages (MB)	
	HPGM	H-HPGM
8	360.7	12.5
12	251.9	9.6
16	193.3	7.8

Table 6: Average amount of received messages on each node

Since H-HPGM sends only the closest to the bottom of large item, the amount of communications is considerably reduced. On the other hand, HPGM has to transmit the itemsets which are generated from the items and their all ancestors, a large amount of communications are caused. Because the performance of HPGM is always much worse than H-HPGM, we omit the performance of HPGM in the following experiments.

4.2 Evaluation of Proposed Algorithms

Figure 14 shows the execution time of all the proposed parallel algorithms, varying the minimum support. 16 nodes in SP-2 are activated in these experiments. The transaction data is evenly spread over the local disks of all the nodes. Although we shows the results at pass 2, the results of the other passes are also very similar to the behavior of pass 2.

In NPGM, the execution time increases sharply when minimum support becomes small. When the candidate itemsets becomes large for small minimum support, the single node's memory cannot hold the entire candidate itemsets. In such a case, NPGM has to divide the candidate itemsets into fragments, each node has to scan the transaction database repetitively for each fragment. Thus the performance of NPGM decreases significantly.

When the minimum support set to be very small, the size of the candidate itemsets becomes large and most of the available memory space is occupied. Thus, the size of the

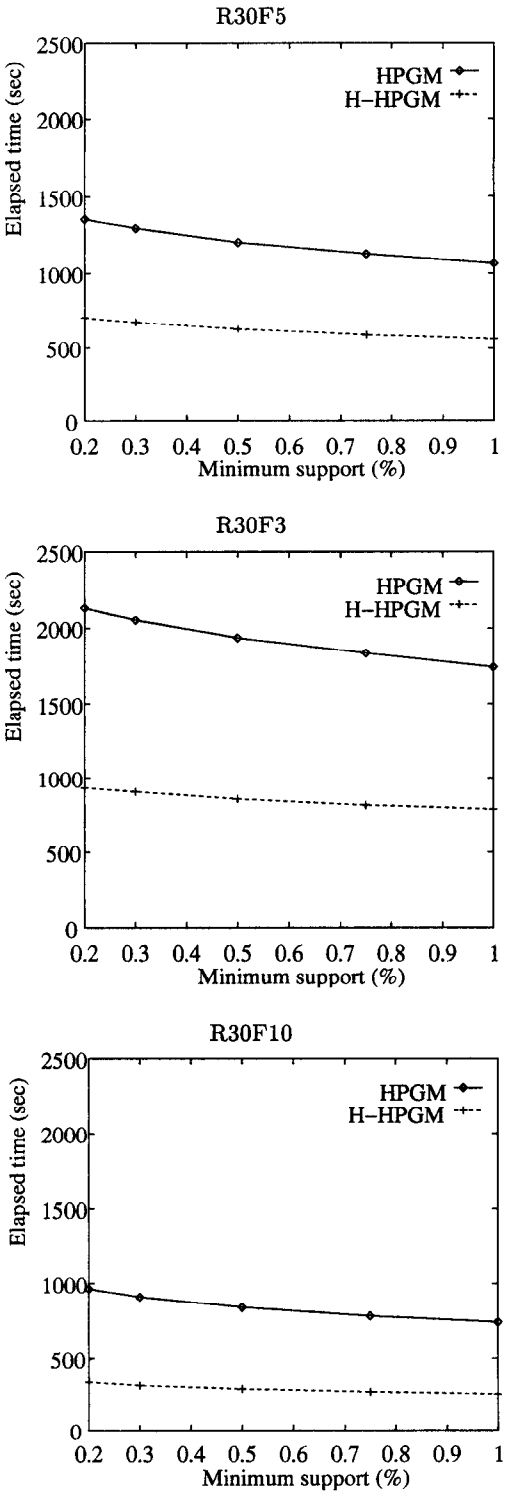

Figure 13: Execution time of HPGM and H-HPGM

Parameter	R30F5	R30F3	R30F10
Number of transactions	3200000	3200000	3200000
Average size of the transactions	10	10	10
Average size of the maximal potentially large itemsets	5	5	5
Number of maximal potentially large itemsets	10000	10000	10000
Number of items	30000	30000	30000
Number of roots	30	30	30
Number of levels	5–6	6–7	3–4
Fanout	5	3	10

Table 5: Parameters of datasets

duplicated candidate itemsets becomes smaller as the minimum support decreases. Since H-HPGM-TGD duplicates the candidate itemsets with a tree grain, it cannot copy the candidate itemsets at small minimum support, where it becomes identical to H-HPGM.

H-HPGM-PGD and H-HPGM-FGD significantly outperform H-HPGM. Since they duplicate the candidate itemsets with small granule, they can more effectively balance the load among the nodes by filling utilizing almost all the free space. Especially, H-HPGM-FGD attain the best performance for all the range of the minimum support. Since H-HPGM-FGD employs the finest grain in our algorithms and utilizes most effectively the free space.

4.3 Comparison of the Workload Distribution

In this section, the workload distribution of H-HPGM, H-HPGM-TGD, H-HPGM-PGD and H-HPGM-FGD is examined. Figure 15 shows the number of hash table probes to increment sup_cou value in each node at pass 2. The minimum support is set to 0.3%. In H-HPGM, the distribution of the number of probe is largely fractured. In H-HPGM-TGD, H-HPGM-PGD and H-HPGM-FGD, we duplicate the frequently candidate itemsets so that the communication can be eliminated.

The grain employed in H-HPGM-TGD is too coarse, it cannot achieve sufficiently flat distribution at small minimum support. H-HPGM-PGD copies the candidate itemsets with path grain, it attains more flat workload distribution than H-HPGM-TGD. However, it sometimes duplicates useless candidate itemsets, since it examines the frequent candidate itemsets at the closest to the leaf level. On the other hand, H-HPGM-PGD can absorb the influence of the transaction data skew effectively, since it duplicates the candidate itemsets in finer grain than H-HPGM-PGD and does not duplicate useless candidate itemsets. This method can must effectively utilize the free space for load balancing.

4.4 Speedup

Figure 16 shows the speedup ratio varying the number of nodes used 4, 6, 8, 12 and 16. The curves are normalized by the 4 nodes execution time. Dataset R30F5 is used. The minimum support is set to 0.5% and 0.3%.

H-HPGM-FGD and H-HPGM-PGD attain higher linearity than H-HPGM. Sophisticated load balancing mechanism contributes to the improvement of linearity. Since H-HPGM does not duplicate the candidate itemsets, data skew significantly degrades the linearity. H-HPGM-TGD, H-HPGM-PGD and H-HPGM-FGD detect the frequent candidate itemsets and copy them among all the nodes. When the available free space is small, H-HPGM-TGD cannot duplicate the candidate itemsets. Thus it does not work well. On the other hand, H-HPGM-FGD employing fine grain duplication strategy can more effectively achieve higher linearity.

5 Conclusions

In this paper, we proposed several kinds of parallel algorithms for mining association rules with classification hierarchy on an shared-nothing parallel machine. We examined their effectiveness through the implementation on 16 node parallel machine.

If a single node can hold all the candidate itemsets, parallelization is straightforward. It is sufficient to partition the transaction over the nodes. Each node can process the allocated transaction data in parallel. We named this algorithm NPGM. However when we generate association rules that span different levels of the classification hierarchy, the candidate itemsets tend to become too large to fit within the memory of a single node. Decreasing a small minimum support also increases the size of the candidate itemsets. As we decrease the minimum support, computation time grows rapidly. But in order to discover more interesting association rules, we usually have to decrease the minimum support, even though we just pick up high confidence rules.

The algorithms, HPGM, H-HPGM, H-HPGM-TGD, H-HPGM-PGD and H-HPGM-FGD not only partition the transaction data file among the nodes but partition the candidate itemsets. HPGM partitions the candidate itemsets without taking the classification hierarchy information into account. It has to exchange both the items that contained in the transaction data and its ancestor items, which causes large amount of communication overhead. H-HPGM partitions the candidate itemsets considering classification hierarchy so that all the candidate itemsets whose root candidate is identical be allocated to the same node. Since H-HPGM transmits only the leaf large items, the amount of communications is considerably reduced. H-HPGM-TGD, H-HPGM-PGD and H-HPGM-FGD detect the frequently candidate itemsets and duplicate them so that the remaining free space can be utilized as much as possible. Support counting for frequent candidate can be locally processed which further reduce the communication overhead a lot.

We implemented these algorithms on a shard-nothing parallel machine 16-node SP-2. Performance evaluations show that H-HPGM-FGD attains sufficiently high performance and achieves almost flat load distribution.

In [SA96], generalized sequential pattern mining with classification hierarchy is discussed. Since generalized sequential pattern mining requires to examine the permutation of items and to include the items across the different levels of classification hierarchy, the size of the candidate sets becomes much larger. In [SK98], we present the parallelization of mining sequential patterns. Extension of our

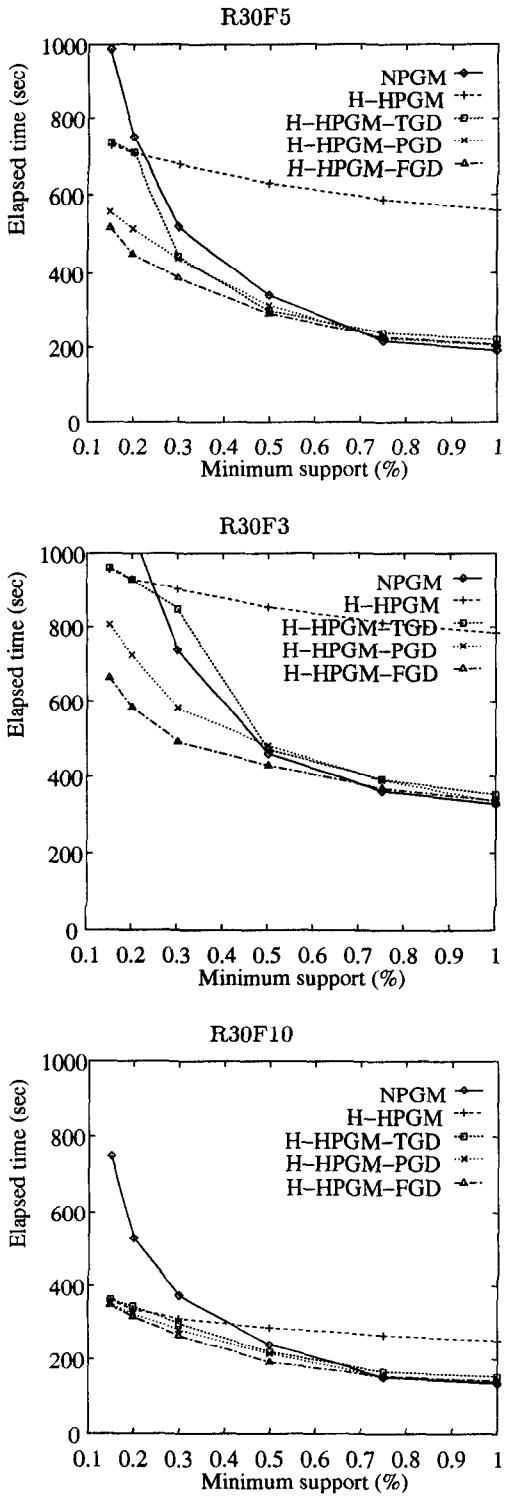

Figure 14: Execution time of HPGM and H-HPGM

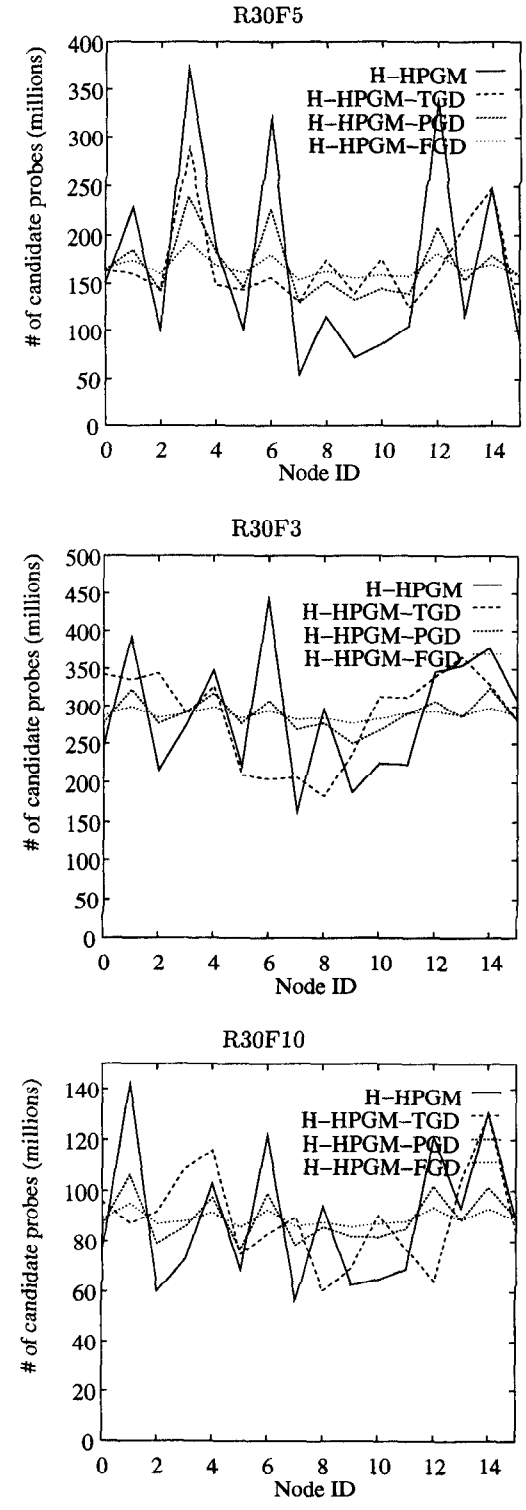

Figure 15: The number of candidate probes for pass 2

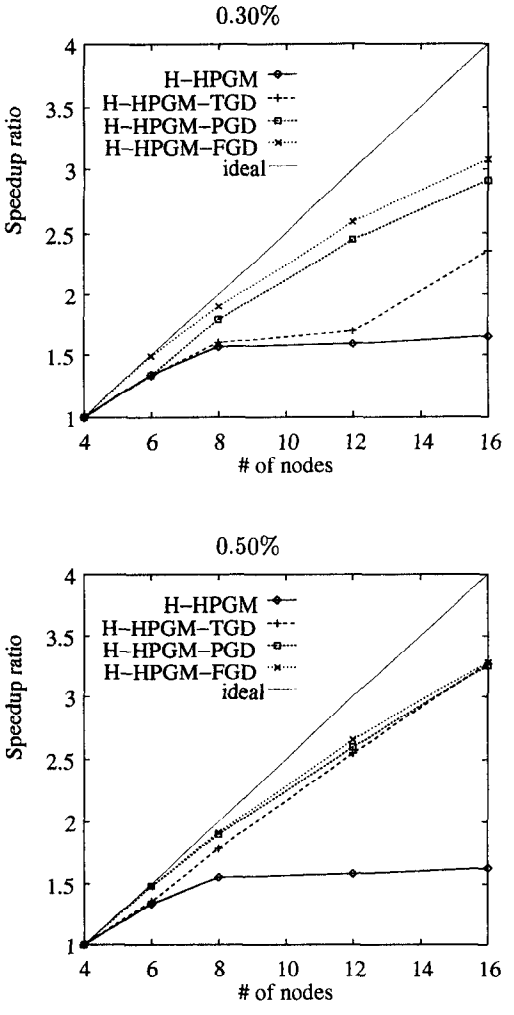

Figure 16: Speedup ratio

parallel algorithms to the mining of generalized sequential patterns is interesting study for future work.

Acknowledgments

This work is supported by the Ministry of Education, Science, Sports and Culture, Japan by the grant of Scientific Research on Priority Areas for " Research and Development of Advanced Database Systems for Integration of Media and User Environments".

References

[AS96] R. Agrawal and J. C. Shafer. Parallel mining of association rules. In *IEEE Transactions on Knowledge and Data Engineering, Vol.8, No.6*, pages 962–969, December 1996.

[CHN+96] D. W. Cheung, J. Han, V. T. Ng, A. W. Fu, and Y. Fu. A fast distributed algorithms for mining association rules. In *Proceedings of IEEE 4th International Conference on Parallel and Distributed Information Systems*, pages 31–42, December 1996.

[CNFF96] D. W. Cheung, V. T. Ng, A. W. Fu, and Y. Fu. Efficient mining of association rules in distributed databases. In *IEEE Transactions on Knowledge and Data Engineering, Vol.8, No.6*, pages 911–922, December 1996.

[HKK97] E. H. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. In *Proceedings of 1997 ACM SIGMOD International Conference on Management of Data*, pages 277–288, 1997.

[PCY95] J. S. Park, M. S. Chen, and P. S. Yu. Efficient parallel data mining for association rules. In *Proceedings of the 4th Conference on Information and Knowledge Management*, pages 31–36, November 1995.

[RR94] R.Agrawal and R.Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499, September 1994.

[SA95] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proceedings of 20th International Conference on Very Large Data Bases*, pages 407–419, September 1995.

[SA96] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proceedings of the 5th International Conference on Extending Database Technology*, March 1996.

[SK96] T. Shintani and M. Kitsuregawa. Hash based parallel algorithms for mining association rules. In *Proceedings of IEEE 4th International Conference on Parallel and Distributed Information Systems*, pages 19–30, December 1996.

[SK98] T. Shintani and M. Kitsuregawa. Mining algorithms for sequential patterns in parallel : Hash based approach. *to be published in the Second Pacific-Asia Conference on Knowledge Discovery and Data mining*, April 1998.