Mining Algorithms for Sequential Patterns in Parallel : Hash Based Approach

Takahiko Shintani and Masaru Kitsuregawa

Institute of Industrial Science, The University of Tokyo
7-22-1 Roppongi, Minato-ku, Tokyo, 106 Japan

Abstract. In this paper, we study the problem of mining sequential patterns in a large database of customer transactions. Since finding sequential patterns has to handle a large amount of customer transaction data and requires multiple passes over the database, it is expected that parallel algorithms help to improve the performance significantly.
We consider the parallel algorithms for mining sequential patterns on a shared-nothing environment. Three parallel algorithms (Non Partitioned Sequential Pattern Mining(NPSPM), Simply Partitioned Sequential Pattern Mining(SPSPM) and Hash Partitioned Sequential Pattern Mining(HPSPM)) are proposed. In NPSPM, the candidate sequences are just copied among all the nodes, which can lead to memory overflow for large databases. The remaining two algorithms partition the candidate sequences over the nodes, which can efficiently exploit the total system's memory as the number of nodes in increased. If it is partitioned simply, customer transaction data has to be broadcasted to all nodes. HPSPM partitions the candidate sequences among the nodes using hash function, which eliminates the customer transaction data broadcasting and reduces the comparison workload. We describe the implementation of these algorithms on a shared-nothing parallel computer IBM SP2 and its performance evaluation results. Among three algorithms HPSPM attains best performance.

1 Introduction

Data Mining, also known as knowledge discovery in databases, has attracted strong attention. Because of the progress of data collection tools, large amount of transaction data have been generated, but such data being archived and not used efficiently. 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 existing data.

The problem of mining sequential patterns in a large database of customer transactions was introduced in [1]. A transaction data typically consists of a customer identifier, a transaction identifier, a transaction time associated with each transaction and the bought items per-transaction. By analyzing these customer-transaction data, we can extract the sequential patterns such as "5% of customer who buy both A and B buy C in the next transaction".

Several algorithms have been proposed to find sequential patterns[1–3]. An algorithm for finding all sequential patterns, named AprioriAll, was presented in [1]. First, AprioriAll discovers all the sets of items (itemset) with a user-specified minimum support (large itemset), where the support is the percentage of customer transactions that contain the itemsets. Secondly, the database is transformed by replacing the itemsets in each transaction with the set of all large itemsets. Lastly, it finds the sequential patterns. It is costly to transform the database. In [3], a graph-based algorithm, DSG(Direst Sequential pattern Generation), was presented. DSG constructs an association graph to indicate the associations between items by scanning the database once, and generates the sequential pattern by traversing the graph. Though the disk I/O cost of DSG is very low, the related information may not fit in the memory when the size of the database is large. In [2], GSP(Generalized Sequential Pattern) algorithm that discovers generalized sequential patterns was proposed. GSP finds all the frequent sequences without transforming the database. Besides, some generalized definitions of sequential patterns are introduced in [2]. First, time constructions are introduced. Users often want to specify maximum and/or minimum time period between adjacent elements. Second, flexible definition of a customer transaction is introduced. It allows a user-specified window-size within which the items can be present. Third, given a user-defined taxonomy (*is-a* hierarchy) over the data items, the generalized sequential pattern, which includes items span different levels of the taxonomy, is introduced. All the previous algorithms for finding sequential patterns are serial algorithms. Finding sequential patterns has to handle a large amount of customer transaction data and requires multiple passes over the database, which requires long computation time. Thus, its computational requirements are too large for a single processor to have a 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)[4]. Association rules are the rules about what items are bought together within a transaction. HPA not only partitions the transaction database but the candidate itemsets among the nodes. Several other parallel algorithms for mining association rules are introduced[5–8]. However, most of these algorithms do not partition the candidate itemsets. Among them, Data Distribution[7], IDD(Intelligent Data Distribution) and HD(Hybrid Distribution)[9] partition the candidate sequences. Since these algorithms exchange all the transaction data among the nodes, a large amount of communications are required. On the other hand, HPA partitions the candidate itemsets using hash function to reduce the amount of communications.

In this paper, we consider the parallel algorithms for mining sequential patterns on a shared-nothing environment. Three parallel algorithms (Non Partitioned Sequential Pattern Mining, Simply Partitioned Sequential Pattern Mining and Hash Partitioned Sequential Pattern Mining) are proposed. In NPSPM, the candidate sequences are just copied among all the nodes. The remaining two algorithms partition the candidate sequences over the nodes, which can efficiently exploit the total system's memory as the number of nodes in increased. If the candidate sequences are partitioned simply, customer transaction data has to be broadcasted to all nodes. HPSPM partitions the candidate itemsets among the nodes using hash function, which eliminates the customer transaction data broadcasting and reduces the comparison workload. We describe the implementation of these algorithms on a shared-nothing parallel computer IBM SP2 and its performance evaluation results. Among three algorithms, HPSPM attains best performance.

The rest of this paper is organized as follows: Section 2 describes the problem of mining sequential patterns. In section 3, we propose three parallel algorithms. Performance evaluations are given in section 4. Section 5 concludes the paper.

2 Sequential Patterns

Here, we introduce some basic concepts of sequential patterns, using the formalism in [2]. Let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of literals, called items. An itemset X is defined a set of items such that $X \subseteq \mathcal{I}$. Let $\mathcal{D} = \{s_1, s_2, \ldots, s_n\}$ be a set of customer-sequences, where each sequence s is a set of transactions ordered by increasing transaction time corresponds to a sequence, $s = \{t_1, t_2, \ldots, t_k\}$. We define t_j an element of the sequence. A transaction t has a set of items ($t \subseteq \mathcal{I}$), customer identifier, transaction identifier and transaction time. We say a sequence $s_a = \langle a_1, a_2, \ldots, a_x \rangle$ contains a sequence $s_b = \langle b_1, b_2, \ldots, b_y \rangle$ if there exist integers $i_1 < i_2 < \ldots < i_y$ such that $a_{i_1} \supseteq b_1, a_{i_2} \supseteq b_2, \ldots, a_{i_y} \supseteq b_y$. The sequence s has support α in the customer-sequence database $\mathcal{D}$, if α% of the customer-sequences in $\mathcal{D}$ contain s. The problem of mining sequential patterns is to find all sequences that satisfy a user-specified minimum support on the assumption that we are given a set of customer-sequences. These sequences are called frequent sequences.

Here, we explain the GSP algorithm for finding all frequent sequences, proposed in [2]. Since we consider finding the naive sequential patterns, several generalized definitions introduced in [2] are ignored. In the first iteration (pass 1), support-count for each item is counted by scanning the customer-sequence database. Hereafter we prepare a field named support-count for each sequences, which is used to measure how many times the sequences contained in customer-sequences. Since sequences here consist of just single item, each item has a support-count fields. All the items which satisfy the minimum support threshold are picked out. These items are called frequent 1-sequences (L_1). Here, k-sequence is defined a sequence with k items. In the second iteration (pass 2), the 2-sequences are generated using L_1. These sequences are called the candidate 2-sequences (C_2). Then the support-count of C_2 is counted by scanning the customer-sequence database. Here support-count of the sequence means the number of customer-sequences which contain it. At the end of scanning the customer-sequence database, the frequent 2-sequences (L_2) which satisfy minimum support are determined. The following denotes the k-th iteration, pass k ($k \geqq 2$).

1. Generate the candidate sequence:
   The candidate k-sequences (C_k) are generated using large $(k-1)$-sequences (L_{k-1}) which were determined in the previous pass (pass $k-1$).
2. Count support:
   The support-count of C_k are counted by scanning the customer-sequence database $\mathcal{D}$.
3. Determine frequent sequence:
   The candidate k-sequences in C_k are checked for whether they satisfy the minimum support condition or not, the frequent k-sequences (L_k) which satisfy the minimum support are determined.

This procedure terminates when the new frequent sequence becomes empty.

The procedure for generating candidate k-sequences is as follows: Candidate generation occurs in two steps. First, in the join step, the candidate k-sequence C_k are generated by joining L_{k-1} with L_{k-1}. If there are the sequences that the subsequence obtained by dropping the first item of s_1 is identical with the subsequence obtained by dropping the last item of s_2, the candidate sequence is generated by adding s_1 to the last item in s_2. The added item becomes a separate element if it was a separate element in s_2, and a part of the last element of s_1 otherwise. Next, in the prune step, delete all of the sequences in C_k where some of the $(k-1)$-subset of the candidate k-sequences are not in L_{k-1}.

3 Parallel Algorithms

In this section, we describe the parallel algorithms for finding all the frequent sequences for shared-nothing parallel machines. In the serial algorithm, the count support processing requires the longest computation time, where the customer-sequence database is scanned and a large number of candidate sequences are examined. We designed a parallel algorithms for the count support processing.

If each node can hold all of the candidate itemsets, parallelization is straightforward. By partitioning the customer-sequence database over all the nodes, disk I/O operation can be done in parallel. In NPSPM, the candidate sequences are just copied among all the nodes. In the case where all of the candidate sequences do not fit within the local memory of single node, the candidate sequences are partitioned into the fragments, each of which fits in the memory size of single node. At this time, NPSPM makes multiple passes over the customer-sequence database in one pass. In order to exploit the total system's memory effectively, the remaining two algorithms partition the candidate sequences over the memory space of all the nodes, and count the support-count by exchanging the customer-sequence among the nodes. For simplicity, we assume that the size of the candidate sequences in one pass is larger than the size of local memory of single node but is smaller than the sum of the memory of all the nodes. It is easy to expand this algorithm to handle the candidate itemsets whose size exceeds the sum of the memory of all the nodes.

3.1 Non Partitioned Sequential Pattern Mining : NPSPM

In NPSPM, the candidate sequences are just copied among all the nodes, each node determine the frequent sequences by exchanging the support count values among all the nodes. Though each node can work independently in count support processing, each node has to examine all the candidate sequences. Figure 1 gives the behavior of pass k of the p-th node, using the notation in Table 1.

Table 1. Notation for NPSPM algorithm

$\mathcal{L}_k$	Sets of all the frequent k-sequences
$\mathcal{C}_k$	Sets of all the candidate k-sequences
$\|\mathcal{C}_k\|$	The size of $\mathcal{C}_k$ in bytes
M	The size of main memory in bytes
$\mathcal{D}^p$	Customer-sequences stored in the local disk of the p-th node
C_k^d	Sets of fragments of the candidate k-sequences. Each fragments fits in the local memory of a node.
$\|C_k^d\|$	The size of C_k^d in bytes
L_k^d	Sets frequent k-sequences derived from C_k^d

$k \geqq 2$
$\mathcal{C}_k :=$ The candidates of size k generated from $\mathcal{L}_{k-1}$
$\{C_k^d\} :=$ Partition $\mathcal{C}_k$ into fragments each of which fits in a node's local memory
$(d=1, \ldots, \lceil |\mathcal{C}_k|/M \rceil)$
for $(d=1;\ d \leqq \lceil |\mathcal{C}_k|/M \rceil;\ d{+}{+})$ **do**
  forall $s \in \mathcal{D}^p$ **do**
    Increment the support_count of all candidates in C_k^d that are contained in s
  end
  Send the support_count of C_k^d to the coordinator node
  /* Coordinator node determine L_k^d which satisfy user-specified minimum sup- */
  /* port in C_k^d and broadcast L_1^d to all the nodes */
  Receive L_k^d from the coordinator node
end
$\mathcal{L}_k := \bigcup_d L_k^d$

Fig. 1. NPSPM algorithm

Each node works as follows:

1. Generate the candidate sequences:
   - Each node generates the candidate k-sequences (C_k) using the large $(k-1)$-sequences (L_{k-1}).
   - Insert C_k into the candidate hash table.
2. Scan the customer-sequence database and count the support-count value:
   - Each node reads the customer-sequence database from its local disk.
   - Increment the support-count of all the candidates in C_k that are contained in s.
3. Determine the large sequences:
   - After reading all the customer-sequences, all node's support-count are gathered into the coordinator node and examined to determine whether the minimum support condition is satisfied or not.

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

If the size of all the candidate sequences exceeds the local memory size of a single node, the candidate sequences are divided into fragments, each of which can fits within the local memory of a single node. Then, the above process is repeated for each fragment. Figure 1, at the beginning of the for-loop, shows the method by which each of the candidate sequences are divided into fragments with each fragment being processed in order.

3.2 Simply Partitioned Sequential Pattern Mining : SPSPM

In NPSPM, the candidate sequences are not partitioned but just copied among all the nodes. However the candidate sequences often becomes too large to fit within the local memory of a single node. SPSPM arbitrarily partitions the candidate sequences equally over the memory space of all the nodes. Thus SPSPM can exploit the aggregate memory of the system. SPSPM assigns the candidate sequences equally to all of the nodes in a round robin manner. By round-robin, the candidates are assigned to the nodes in a cyclical manner with the i-th candidate assigned to node $i \bmod n$, where n is the number of nodes in the sysytem. Since SPSPM partitions the candidate sequences among the nodes, each node has to broadcast the customer-sequences stored in its local disk to all the other nodes for count support processing, while NPSPM requires no such broadcast. Figure 2 shows the behaviour of pass k by the p-th node in SPSPM, using the notation in Table 2. Here we assume the size of candidate itemset is smaller than the size of sum of all the processor's memory. Extension of the algorithm to handle much larger candidate itemset is easy. We can divide the candidate sequences into fragments like in NPSPM.

Table 2. Notation for SPSPM algorithm

$\mathcal{L}_k$	Sets of all the large k-sequences.
$\mathcal{C}_k$	Sets of all the candidate k-sequences.
$\mathcal{D}^p$	Customer-seqnences stored in the local disk of the p-th node
C_k^p	Sets of the candidate k-sequences assigned the p-th node
L_k^p	Sets of large k-sequences derived from C_k^p

Each node works as follows:

1. Generate the candidate sequences:
   – Each node generates $\mathcal{C}_k$ using $\mathcal{L}_{k-1}$ and inserts a part of the candidate itemsets into its own hash table (C_k^p). The candidate k-sequences are assigned to nodes in a round-robin manner.
2. Scan the customer-sequence database and count the support-count value:
   – Each node reads the customer-sequence database $\mathcal{D}^p$ from its local disk and broadcasts them to all the nodes. For each customer-sequence entry, the support-count is incremented in the same way as in NPSPM, when read from its own disk or received from another nodes.

$k \geqq 2$
$\mathcal{C}_k$:=The candidates of size k generated from $\mathcal{L}_{k-1}$
$\{C_k^p\}$:=All the candidate k-itemsets, assigned to the p-th node
forall $s \in \mathcal{D}^p$ **do**
    Broadcast s to all the nodes
    Increment the support-count value of C_k^p contained in s
    Receive the customer-sequence s' sent from the other nodes and increment the support-count of all candidates that are contained in s'
end
$\{L_k^p\}$:=All the candidates in C_k^p with minimum support
Send L_k^p to the coordinator node
/* Coordinator node make up $\mathcal{L}_k := \bigcup_p L_k^p$ and broadcast to all the nodes */
Receive $\mathcal{L}_k$ from the coordinator node

Fig. 2. SPSPM algorithm

3. Determine the large sequences:
   – After reading all the customer-sequences, each node can determine individually whether each candidate in C_k^p satisfy minimum support condition or not.
   – Each node send L_k^p to the coordinator node, where all the large k-sequences $\mathcal{L}_k := \bigcup_p L_k^p$ are derived.
4. If $\mathcal{L}_k$ is empty, the algorithm terminates. Otherwise the coordinator node broadcasts $\mathcal{L}_k$ to all the nodes, $k := k + 1$ and goto "1".

Example 1. Assuming there are 3 nodes in a system. Let the large items, computed pass 1, be $L_1 = \{1, 2, 3, 5, 7\}$. Since SPSPM partitions the candidate sequences in round-robin manner, the candidate sequences are assigned shown in Figure 3. Suppose Node 0 reads the customer-sequence $s = \{1, 3, 6\}\{2, 4\}$. Then Node 0 broadcast s to all the other nodes and increment the count-support of $\{1, 3\}, \{3\}\{2\}$. Node 1 and Node 2 receive the sequence $\{1, 3, 6\}\{2, 4\}$ from Node 0. Node 2 increments the support-count of $\{1\}\{2\}$.

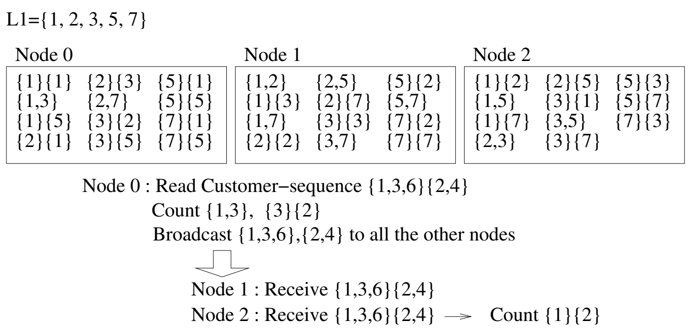

Fig. 3. An example of SPSPM algorithm

3.3 Hash Partitioned Sequential Pattern Mining : HPSPM

In SPSPM, the amount of communications becomes very large, since SPSPM requires broadcasting all the customer-sequences in the count support processing. HPSPM partitions the candidate sequences among the nodes using the hash function like in the hash join in relational database systems, which eliminates the need for customer-sequence broadcasting and can reduce the comparison workload significantly. Figure 4 gives the behavior of pass k by the p-th node, using the notation in Table 2.

$k \geqq 2$
$\mathcal{C}_k$:=The candidates of size k generated from $\mathcal{L}_{k-1}$
$\{C_k^p\}$:= All the candidate k-itemsets, whose hashed value corresponding to the p-th node
forall $s \in \mathcal{D}^p$ **do**
  s' := Select all the items that contained in $\mathcal{C}_k$ from the customer-sequence s
  forall k-sequence $x \in s'$ **do**
    Determine the destination node ID by applying the same hash function which is used in candidate sequences partitioning, and send that k-sequences to it. If it is its own ID, increment the support-count for the sequence.
    Receive k-sequences from the other nodes and increment the support-count for that sequence
  end
end
$\{L_k^p\}$:=All the candidates in C_k^p with minimum support
Send L_k^p to the coordinator node
/* Coordinator node make up $\mathcal{L}_k := \bigcup_p L_k^p$ and broadcast to all the nodes */
Receive $\mathcal{L}_k$ from the coordinator node

Fig. 4. HPSPM algorithm

Each node works as follows:

1. Generate the candidate sequence:
   - Each node generates $\mathcal{C}_k$ using $\mathcal{L}_{k-1}$.
   - 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 candidate hash table (C_k^p).
2. Scan the customer-sequence database and count the support-count value:
   - Each node reads the customer-sequence database from its local disk, and generates new customer-sequence s' by selecting all the items that are contained in $\mathcal{C}_k$ from the customer-sequence s.
   - Generate k-sequences from s' and apply the same hash function used in phase 1. Derive the destination node ID and send the k-sequence to it.
   - For the sequences received from other nodes and those locally generated whose ID equals the node's own ID, search the candidate hash table. If hit, increment its support-count value.
3. – 4. Same as SPSPM

In Figure 5, Example 2 illustrates the behavior of HPSPM for the same condition at Example 1.

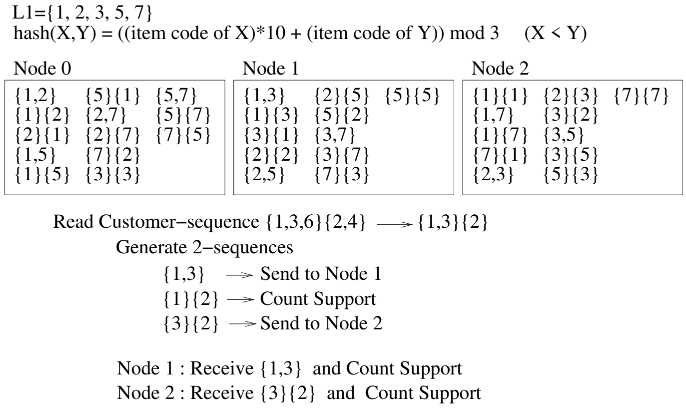

Fig. 5. An example of HPSPM algorithm

Example 2. Let the hash function be $hash(X, Y) = ((item\ code\ of\ X) * 10 + (item\ code\ of\ Y))\ mod\ 3$, here $X < Y$. Therefor, the candidate sequence $\{1, 2\}$ whose hashed value is 0 is allocated to Node 0. The other candidate sequences are allocated the same way. Suppose Node 0 reads the customer-sequence $s = \{1, 3, 6\}\{2, 4\}$. Then Node 0 generates new customer-sequence $s' = \{1, 3\}\{2\}$ by selecting the large items in s, generates 2-sequences from s' and applies the hash function used at the candidate sequence partitioning. For example, since the hash value of $\{3\}\{2\}$ is 2, Node 0 sends $\{3\}\{2\}$ to Node 2. Node 2 receives $\{3\}\{2\}$ from Node 0 and increments the support-count of $\{3\}\{2\}$.

4 Performance Evaluation

We measured all the experiments on a 16-node IBM SP2, which employs a shared-nothing architecture. Each node is a POWER2 with 256MB local memory and a 2GB local disk drive. Each node communicates with each other through HPS (High-Performance Switch).

To evaluate the performance of the proposed parallel algorithms, synthetic datasets emulating retail transactions are used, where the generation procedure is based on the method described in [2]. Table 3 shows the meaning of the various parameters and the characteristics of the datasets used in our experiments. In the following, we use the notation CxTy to denote a dataset in which an average number of transactions per customer-sequence is x and an average number of items per transaction is y.

4.1 Measurement of Execution Time

Figure 6 shows the execution time of the proposed parallel algorithms for pass 2, varying the minimum support condition. The customer-sequence database is evenly spread over the node's local disks before the experiments starts. In these

Table 3. Parameters of datasets

Parameter	C10T5	C20T2.5
Number of customer-sequences (= size of Database)	384,000	384,000
Average number of transactions per customer-sequence	10	20
Average number of items per transaction	5	2.5
Average length of maximal potentially frequent sequences	4	4
Average size of Itemsets in maximal potentially frequent sequences	1.25	1.25
Number of maximal potentially frequent sequences	5,000	5,000
Number of maximal potentially frequent itemsets	25,000	25,000
Number of items	10,000	10,000

experiment, we measure the performance only for pass 2. Since the single node's memory cannot hold the entire candidate sequences only for pass 2.

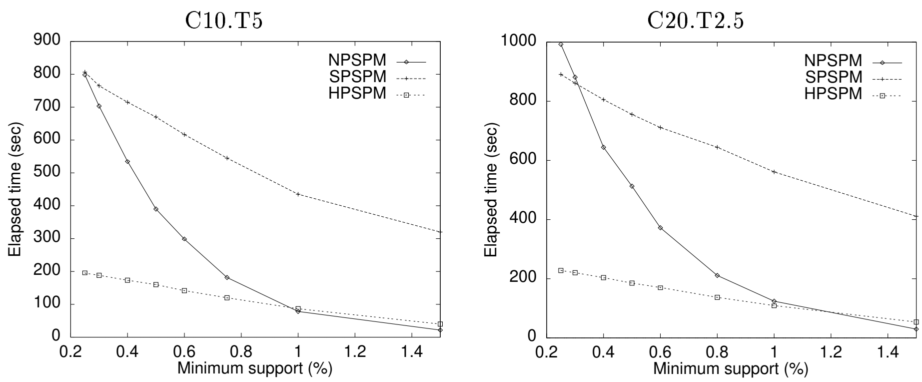

Fig. 6. Execution time for pass 2

In NPSPM, the execution time increases sharply when minimum support becomes small. When the minimum support is small, the size of the candidate sequences becomes large and the single node's memory cannot hold the entire candidate sequences. In such case, NPSPM divides the candidate sequences into fragments, and scans the customer-sequence database repetitively for each fragment. Thus the execution time of NPSPM increases sharply as decreases the minimum support threshold.

HPSPM always outperforms SPSPM. In HPSPM and SPSPM, each node exchanges the customer-sequence data among the nodes for support count processing, since these two algorithms partition the candidate sequences over the memory space of all the nodes. Since SPSPM partition the candidate sequences arbitrarily among the nodes, each node has to broadcast its local customer-sequence to all the other nodes, which causes a large amount of communications. On the other hand, since HPSPM partitions the candidate sequences using hash function, each node transmits the sequences which are generated from the items in a customer-sequence to certain node, the communication costs is reduced.

4.2 Speedup

Figure 7 shows the speedup ratio for pass 2. Here, the number of nodes are varying 4, 6, 8, 12 and 16. The curve is normalized with the execution time on 4 nodes. The minimum support threshold was set to 0.75% for C10T5 and 0.8% for C20T2.5.

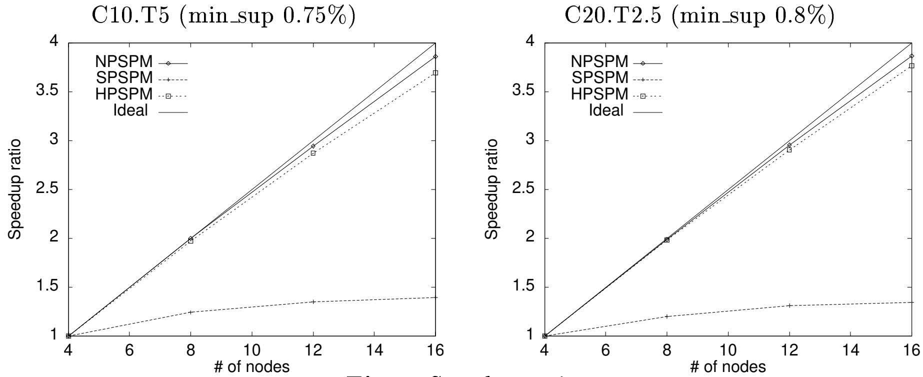

Fig. 7. Speedup ratio

HPSPM and NPSPM attain much higher linearity than SPSPM. Since SPSPM has to broadcast the customer-sequence data to all the other nodes, the communication cost of SPSPM is very high. On the other hand, HPSPM transmits the sequences to certain node, HPSPM can reduce the communication cost. Since NPSPM requires no customer-sequence data transfer for count support processing, NPSPM can attain good linearity. However, the candidate sequences are divided and the support-counts are counted by scanning the customer-sequence database repeatedly in the case the entire candidate sequences do not fit within the memory space of a single node, and each node works on the entire candidate sequences. As a result, the response time of NPSPM is long. Figure 7 show that HPSPM can attain comparable good linearity.

5 Conclusions

In this paper, we proposed three different parallel algorithms for mining sequential patterns on a shared-nothing parallel machine. Since mining sequential patterns requires multiple passes over the customer-sequence database and the database to be mined are often very large, parallel algorithms are required.

In NPSPM, the candidate sequences are just copied over all the nodes, no customer-sequence data transfer for count support processing. However, if the entire candidate sequences do not fit with in the memory space of a single node, the candidate sequences are divided to fragments and the support-counts are counted by scanning the customer-sequence database repeatedly. Since the size of the customer-sequence database is very large, it is very costly to scan the database repeatedly for each fragment. The remaining two algorithms, SPSPM and HPSPM, not only partition the customer-sequence database but partition the candidate sequences among all the nodes, which can exploit the total system's memory effectively, and count the support-count of the candidate sequences by transmit the customer-sequence data. Since SPSPM partition arbitrarily partition the candidate sequences among the nodes, each node has to broadcast its local customer-sequence data to every other nodes, a large amount of communications is caused. HPSPM partitions the candidate sequences using hash function, which avoids the customer-sequence data broadcasting. Consequently, the comparison system workload is reduced significantly. We evaluated the performance of the three algorithms by implementing them on 16-nodes shared-nothing parallel machine, and showed HPSPM can attain relatively good performance.

In [2], generalized sequential pattern is introduced. Since generalized sequential patterns include the items across different levels of the taxonomy over the data items, we have to examine the associations between the different levels of the taxonomy in such mining. Thus, the size of the candidate sequences becomes much larger than mining naive single level sequential patterns. Extension of our parallel algorithms to the mining of generalized sequential pattern is interesting study for future work.

Acknowledgment

This work is (partially) supported as the Grant-in-Aid for Creative Basic Research #09NP1401: "Research on Multimedia Mediation Mechanism for Realization of Human-oriented Information Environments" by the Ministry of Education, Science, Sports and Culture.

References

1. R.Agrawal and R.Srikant. Mining sequential patterns. In *Proc. of the 11th Int. Conf. on Data Engineering*, pages 3–14, March 1995.
2. R.Srikant and R.Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. of the 5th Int. Conf. on Extending Database Technology*, March 1996.
3. S.-J.Yen and A.L.P.Chen. An efficient approach to discovering knowledge from large databases. In *Proc. of 4th Int. Conf. on Parallel and Distributed Information Systems*, pages 8–18, December 1996.
4. T.Shintani and M.Kitsuregawa. Hash based parallel algorithms for mining association rules. In *Proc. of 4th Int. Conf. on Parallel and Distributed Information Systems*, pages 19–30, December 1996.
5. J.S.Park, M.-S.Chen, and P.S.Yu. Efficient parallel data mining for association rules. In *Proc. of the 4th Conf. on Information and Knowledge Management*, pages 31–36, November 1995.
6. D.W.Cheung, J.Han, V.T.Ng, A.W.Fu, and Y.Fu. A fast distributed algorithms for mining association rules. In *Proc. of 4th Int. Conf. on Parallel and Distributed Information Systems*, pages 31–42, December 1996.
7. R.Agrawal and J.C.Shafer. Parallel mining of association rules. In *IEEE Trans. on Knowledge and Data Engineering, Vol.8, No.6*, pages 962–969, December 1996.
8. D.W.Cheung, V.T.Ng, A.W.Fu, and Y.Fu. Efficient mining of association rules in distributed databases. In *IEEE Trans. on Knowledge and Data Engineering, Vol.8, No.6*, pages 911–922, December 1996.
9. E.-H.(Sam)Han, G.Karypis, and V.Kumar. Scalable parallel data mining for association rules. In *Proc. of 1997 ACM SIGMOD Int. Conf. on Management of Data*, pages 277–288, 1997.