2004－MPS－50　(3)
2004／6／22

並列 Modified PrefixSpan 法における動的負荷分散手法

高木　允[†]　田村慶一[††]
周藤俊秀[†]　北上始[††]

モチーフはアミノ酸配列中に存在する特徴的なパターンであり，生物学的に意味があると考えられている．アミノ酸配列中のモチーフを効率的に発見するために，高速な頻出パターン抽出アルゴリズムが求められている．本稿では，PC クラスタ上でアミノ酸配列から頻出パターンを並列に抽出する並列 Modified PrefixSpan 法の動的負荷分散手法を示す．並列 Modified PrefixSpan 法は PC クラスタ間でタスクを分配する手法を用いている．我々は，並列 Modified PrefixSpan 法の動的負荷分散手法としてマスタ・タスク・スティル法を示す．マスタ・タスク・スティル法は PC クラスタ上で各 PC の負荷が偏っている場合，負荷が少ない PC の空き時間を最小限にする手法である．

Dynamic Load Balancing for Parallel Modified PrefixSpan

Makoto Takaki,[†] Keiichi Tamura ,[††] Toshihide Sutou[†]
and Hajime Kitakami[††]

Motif is featured pattern which is biologically meaningful in amino acid sequences. Motif is discovered from frequent patterns. In order to extract the frequent patterns that can become motifs in amino acid sequences at high-speed, a parallel Modified PrefixSpan is proposed. This paper presents a dynamic load balancing for the parallel Modified PrefixSpan to extract frequent patterns at high-speed from the amino acid sequences on a PC cluster. The parallel Modified PrefixSpan exploits a task-based parallelism that distributes tasks among the computers on the PC cluster. We present a dynamic load balancing methodology called master-task-steal. The master-task-steal-based dynamic load balancing minimizes idle time when the distributed workload is unbalanced on the PC cluster.

1. Introduction

Motifs in biologically are assumed to be related to a function of proteins that have been preserved in the evolutionary process of an organism. The Motif is discovered from frequent patterns in amino acid sequences.

In order to discover the motifs efficiently, a Modified PrefixSpan (called MPS)[6] with a rich functional frequent pattern extraction algorithm is proposed. The MPS can extract the frequent patterns including fixed-length wildcard regions and existing in different positions in amino acid sequences.

† 広島市立大学大学院情報科学研究科
Graduate School of Information Sciences, Hiroshima City University
†† 広島市立大学情報科学部
Faculty of Information Sciences, Hiroshima City Unievrsity

We have been developing a parallel Modified PrefixSpan (called PMPS)[10] to extract the frequent patterns at high-speed on a PC cluster. The PMPS exploits a task-based parallelism which distributes tasks among the computers on the PC cluster (a computer on the PC cluster is called a site) and master-worker parallelism.

This paper presents a dynamic load balancing for the PMPS. The characteristic of the dynamic load balancing for the PMPS is a master-task-steal (MTS) methodology. The MTS-based dynamic load balancing is a method of declustering tasks from some working sites to sites that have finished all task. The MTS methodology minimizes idle time when the distributed workload is unbalanced on the PC cluster.

We evaluated the PMPS with the MTS-based dynamic load balancing on an actual PC cluster. In the experiments, two types of data sets that in-

clude motifs named Zinc Finger and Kringle were used. The speedup of the PMPS with the MTS-based dynamic load balancing is superior to that of the previous implementation of the PMPS.

The rest of the paper is organized as follows: Section 2 discusses related work. Section 3 explains the MPS. Section 4 describes the PMPS and the MTS-based dynamic load balancing technique. Section 5 shows the experiment results. Section 6 is the conclusions.

2. Related Work

There are many studies on frequent pattern extraction in sequence databases. Most of these studies adopt, an *apriori like*[3], a candidate generation-and-test approach. The *apriori like* approach may still be expensive, especially when long and numerous patterns are encountered. In order to extract frequent patterns, a new methodology, called *frequent pattern growth (FP-Growth)*, was developed by Jiawei Han and Jian Pei[5]. In this approach, a divide-and-conquer philosophy is used to project and partition databases based on the currently discovered frequent patterns and grow such patterns into longer ones in the projected databases. This approach mines the frequent patterns without candidate generation.

The proposed *apriori like* or *FP-Growth* frequent pattern extraction algorithms focus on business field data, such as market basket data[9], episode data[4], and network log data. These algorithms are not adapted to bioinformatics field data. The MPS is proposed to extract frequent patterns in amino acid sequences. The MPS is an extension of the PrefixSpan[7], which is based on the *FP-Growth* approach. The key idea of the MPS is to examine only prefix subsequences and to project only their corresponding postfix subsequences into projected databases.

To the best of our knowledge, there is no study on the parallel processing of *FP-Growth* frequent pattern extraction algorithms. There are many studies on the parallel processing of the *apriori like* approach[2)8)11)]. However, these parallelisms are not adapted to the *FP-Growth* approach. The main cost of the *apriori like* approach is candidate generation and scanning sequence databases. The main cost of the *FP-Growth* approach is the construction of frequent patterns from postfix databases and the generation of the projected database of frequent patterns.

3. Modified PrefixSpan

3.1 Problem Definition

In this section, we first define the problem of frequent pattern extraction in the protein sequences. Let Σ = {A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y } be a set of all letter alphabet in the amino acid sequences. Sequence s is denoted as $(a_1 a_2 \cdots a_m)$, where a_j is a letter, i.e., $a_j \in \Sigma$, $a_j = s[j]$, for $1 \leq j \leq m$. Sequence database S is a set of tuples $\langle sid, s_{sid} \rangle$, where sid is a sequence id and s_{sid} is a sequence.

A k-length pattern is denoted as $pat^k = \langle A_1 - x(i_1) - A_2 - x(i_2) - \cdots - x(i_{k-1}) - A_k \rangle$. Symbol A_j is called a character element. Symbol $(-)$ means that the next element is continued. Symbol $x(i_n)$ represents fixed-length wildcard regions (where $0 \leq i_n \leq max\_wc$, the maxium length of a fixed-length wildcard region in the frequent patterns is denoted as max_wc).

The support of the k-length pattern pat^k in S is the number of tuples in S containing pat^k. Given a positive integer, min_sup as the support threshold, the k-length pattern is called a k-frequent (sequential) pattern if pat^k is contained by at least min_sup tuples. The k-frequent pattern pat^k is represented as "$pat^k : cnt$", if the support of pat^k is cnt. Supposing that there are n k-frequent patterns extracted from S, these are denoted as P_k = { $\langle pat_1^k \rangle : cnt_1$,$\langle pat_2^k \rangle : cnt_2$, $\cdots$, $\langle pat_n^k \rangle : cnt_n$ }.

Each pat^k has a projected database that keeps next positions of the rightmost characters of pat^k. It is denoted as $PDB(pat^k) = \{(sid, pos) |$ pos is the next position of the rightmost character of pat^k, where $1 \leq pos \leq \| s_{sid} \|$ }.

3.2 MPS Algorithm

Fig.1 shows the algorithm of MPS.

- **phase 1:**
  The algorithm scans the sequence database S once to find all 1-frequent patterns in the sequences. Each 1-frequent pattern has a

```
MPS(min_sup,max_wc)
PDBLIST :=φ ;
/* PDBLIST is a set of projected database */
create P1 and PDBLIST by scanning S;
while (PDBLIST ≠ φ )
  NEXT_PDBLIST:=φ ;
  for all PDB(pat^k) ∈ PDBLIST
  /*  k is positive integer */
    NEXT_PDBLIST :=
      NEXT_PDBLIST∪ BMPS(min_sup,max_wc,PDB(pat^k));
  end_for;
  swap(PDBLIST,NEXT_PDBLIST);
end_while;
end;

subroutine BMPS(min_sup,max_wc,PDB(pat^k))
TMP_PDBLIST :=φ ; NEW_PDBLIST :=φ ;
for w = 0 to max_wc
  for all (i,j) ∈ PDB(pat^k)
     pat^{k+1} :=
      pat^k ⓒ-x(w)-ⓒ s_i[j+w];
      /*ⓒ means concatenation*/
    if PDB(pat^{k+1}) ∋ TMP_PDBLIST then
      /*∋ means not inclusion*/
      TMP_PDBLIST:=TMP_PDBLIST∪PDB(pat^{k+1});
    end_if;
      PDB(pat^{k+1}) :=
       PDB(pat^{k+1})∪(i,j+w+1);
  end_for;
end_for;
for all PDB(pat^{k+1}) ∈ TMP_PDBLIST
  if support(PDB(pat^{k+1})) ≥ min_sup then
    NEW_PDBLIST:=NEW_PDBLIST∪PDB(pat^{k+1});
    P_{k+1} := P_{k+1}∪ <pat^{k+1}>:support(PDB(pat^{k+1}));
  end_if;
end_for;
return NEW_PDBLIST;
end;
```

Fig.1 MPS Algorithm

Table 1 Example of Amino Acid Sequences

sequence id	sequence
1	MFKALRTIPVILNMNKDSKLCPN
2	MSPNPTNHTGKTLR

$PDB(pat^1)$. In addition, the $PDB(pat^1)$ is inserted into $PDBLIST$.

- **phase 2:**
  For each $PDB(pat^k)$ (where $k \geq 1$) in the $PDBLIST$, the algorithm constructs $(k+1)$-length patterns based on $PDB(pat^k)$. If the support of a $(k+1)$-length pattern is more than min_sup, the $(k+1)$-length pattern is a $(k+1)$-frequent pattern. Each $(k+1)$-frequent pattern has a $PDB(pat^{k+1})$. If no $(k+1)$-frequent pattern is extracted, the algorithm terminates the frequent pattern extraction. Otherwise, $k := k+1$ and go to "phase2";

3.3 Example

Let our running database be a sequence database S given in **Table 1**. Each parameter is as follows:

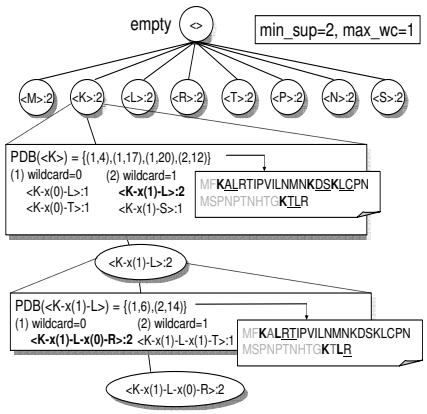

Fig.2 Example of Pattern Growth Steps in the MPS

$min\_sup = 2$, $max\_wc = 1$. The 1-frequent patterns are like P_1=$\{\langle M\rangle : 2, \langle K\rangle : 2, \langle L\rangle : 2, \langle R\rangle : 2, \langle T\rangle : 2, \langle P\rangle : 2, \langle N\rangle : 2, \langle S\rangle : 2\}$. In **Fig.2**, the projected database of 1-frequent pattern $\{\langle K\rangle : 2\}$ is represented by $PDB(\langle K\rangle)$ =$\{$(1,4), (1,17), (1,20), (2,12)$\}$. First, letters $s_1[20+0] = \langle L\rangle$ and $s_2[12+0] = \langle T\rangle$ are appended to $\langle K - x(0) -\rangle$ if the number of wildcards is 0. The 2-length patterns $\{\langle K - x(0) - L\rangle : 1, \langle K - x(0) - T\rangle : 1\}$ are not 2-frequent patterns. Next, if the number of wildcards is 1, letters $s_1[4+1] = \langle L\rangle$, $s_1[17+1] = \langle S\rangle$, and $s_2[12+1] = \langle L\rangle$ are appended to $\langle K - x(1) -\rangle$. Pattern $\langle K - x(1) - L\rangle$ is a 2-frequent pattern because the support of $\langle K - x(1) - L\rangle$ is more than min_sup. The 3-frequent pattern $\langle K - x(1) - L - x(0) - R\rangle$ is extracted in the same way.

4. PMPS with MTS-based Dynamic Load Balancing

4.1 Master-Worker Parallelism

The PMPS (parallel Modified PrefixSpan) is based on the master-worker parallelism. There are two types of processes in the PMPS: the master process and the worker process. The master and worker processes work as follows.

Master Process:

(1) First of all, the master process extracts 1-frequent patterns to s-frequent patterns where s is a user-specified threshold. The process, which extracts all k-frequent patterns (where

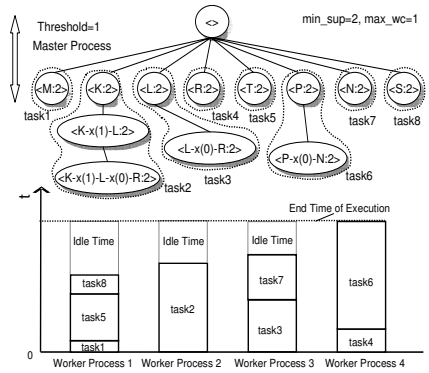

Fig.3 Example of Parallel Pattern Growth Steps in the PMPS

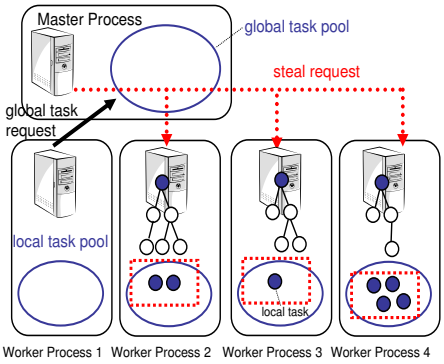

Fig.4 Conceptual Figure of Master-Task-Steal Methodology

$k > s$) from an s-frequent pattern, is called a task. Each task is inserted into a *global task pool*.

(2) The master process sends a task to a worker process when the mater process receives the *global task request* from the worker process.

(3) If the *global task pool* is not empty, the master process returns to process (2). Otherwise, the master process sends a termination signal to all worker processes and terminates processing.

Worker Process:

(1) The worker process sends a *global task request* to the master process.

(2) If the worker process receives a task from the master process, the worker process extracts all k-frequent patterns (where $k > s$) from an s-frequent pattern and returns to process (1). If the worker process receives the termination signal, the worker process sends all results to the master process and terminates processing.

4.2 Dynamic Load Balancing

4.2.1 Master-Task-Steal Methodology

The execution time becomes increasingly unbalanced when the workload of an assigned task at each worker process is very unbalanced. A worker process with extremely big tasks has to continue the processing of tasks with one process though other worker processes have finished the processing of tasks.

For example, as shown in **Fig.3**, the master process generates eight tasks (where threshold=1). Each task is assigned to each site. Finally, each worker process that has finished the processing of tasks, namely, *worker process1*, *worker process2*, and *worker process3* waits for *worker process4*. This wait generating idle time in each worker process. Therefore, an effective speedup ratio cannot be achieved.

The task scheduling algorithms are not adapted to the PMPS. The processing of the task can be done independently. However, it is impossible to estimate the cost of the task because the cost of the task depends on the feature of amino acid sequences.

To overcome this performance limitation on the PMPS, a master-task-steal (MTS)-based dynamic load balancing technique is presented. A key idea of the MTS-based dynamic load balancing is that the balance of the workload on all sites is kept by on-demand declustering distributed tasks. In the MTS methodology, the master process gathers all tasks located in each worker process if the *global task pool* is empty when the master process receives a *global task request*.

In Fig.3, *worker process1* is the first to finish processing the task. *Worker process1* sends a *global task request* to the master process. However, the *global task pool* is empty, and there are no tasks to distribute. In the MTS approach, master process sends a *steal request* to all the worker processes. In **Fig.4**, the worker process, which has received the *steal request*, sends all tasks located in the *local task*

pool to the *global task pool*. There are seven gathered tasks in this case. One of these tasks is then assigned to *worker process 1*.

4.2.2 Master-Task-Steal Algorithm

This section describes an algorithm of MTS-based dynamic load balancing. The processes shown in Section 4.1 are changed as described in the follows.

The worker process extracts from $(s+1)$-frequent patterns to $(s+s'+1)$-frequent patterns where s' is a user-specified threshold. The process that extracts all k-frequent patterns (where $k > (s+s'+1)$) from an $(s+s'+1)$-frequent pattern is called a local task.

Each worker process has its own *local task pool* which can store the tasks. Each local task is inserted into the *local task pool*. The worker process pops a local task from the *local task pool* and executes the local task. The worker process sends a *global task request* to the master process if the *local task pool* is empty.

The master process sends a *steal request* as a broadcast message to all worker process if the *global task pool* is empty. The worker process sends the local tasks to the master process when the worker process receives the *steal request* from the master process. The master process inserts all received local tasks into the *global task pool*. Then the master process re-distributes them to the worker processes.

5. Performance Evaluation

We implemented the PMPS with the MTS-based dynamic load balancing on an actual PC cluster. There are 16 personal computers, each configured with a 2.53GHz Pentium4 processor, 1.5GB memory, and 80GB disk. The personal computers were connected to a 100 Mbit/sec Ethernet. RedHat9.0 was used as the operating system. MPICH version 1.2.5 was used as the MPI library, and GNU g++ ver.3.2.2 was used as the C++ compiler.

The data sets used in this evaluation were provided by PROSITE[1)]. The data set which includes motif named Kringle has 70 data records (total-length: 23,385 bytes; average-length: 334 bytes; maximum-length: 3176 bytes; and minimum-length: 53bytes). The data set which includes motif named Zinc Finger has 467 data records (total-length: 245,595 bytes; average-length: 525 bytes; maximum-length: 4036 bytes; and minimum-length: 34bytes).

For this experiment, each parameter was as follows: Kringle; min_sup=14 (20%); max_wc=5. Zinc Finger; min_sup=140 (30%); max_wc=5.

First, we use dataset including kringle. **Fig.5** shows the speedup ratios of the previous implementation of the PMPS and **Fig.6** shows the speedup ratios of the PMPS with the MTS-based dynamic load balancing. The threshold is denoted as $Threshold(s, s')$, where s and s' is a user-specified threshold of the master process and threshold of the worker processes. Fig.5 shows that enlarging the threshold of the master process increases the speedup ratio to the ideal. It is because the grain of the task became fine and the workload of each worker process became uniform. When the threshold of the master process is 5, the speedup ratio is lower than when it is 4. The serial processing in the master process becomes the bottleneck. As shown in Fig.6, when the thresholds were $Threshold(1, 1)$, the speedup ratio was lower than the other speedups because the grain of the tasks was not fine enough. If the grain of the tasks is rough, the tasks for redistribution decrease. Otherwise, the speedup ratio obtained was 15 times higher than that obtained using just one processor.

Fig.7 shows the execution time of each worker process without MTS-dynamic load balancing when 16 machines are used ($Threshold(1, 0)$). The entire execution time depends on *worker process 2*. The idle time is generated in other worker processes. **Fig.8** shows the execution time of each worker process with MTS-dynamic load balancing when 16 machines are used ($Threshold(1, 3)$). The execution time of each worker process is almost equal, and the uniformity of workloads can be confirmed.

Fig.9 shows the communication frequency and amount of communication. When MTS-basd dynamic load balancing is integrated the maximum total amount of communication is about 1.8Mbytes. Such communication hardly influences the execution time.

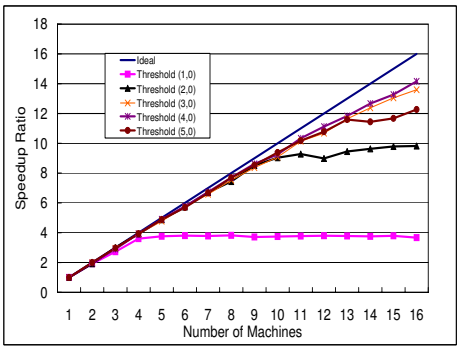

Fig.5 Speedup without MTS-Based Dynamic Load Balancing (Kringle)

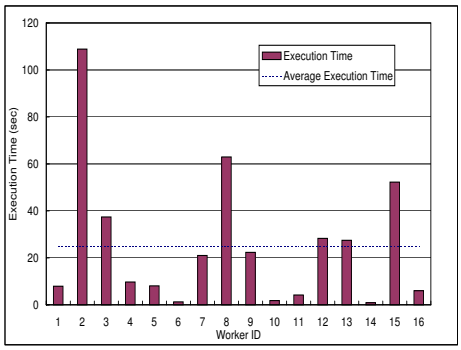

Fig.7 Execution Time of Each Worker Process without MTS-Based Dynamic Load Balancing (Kringle)

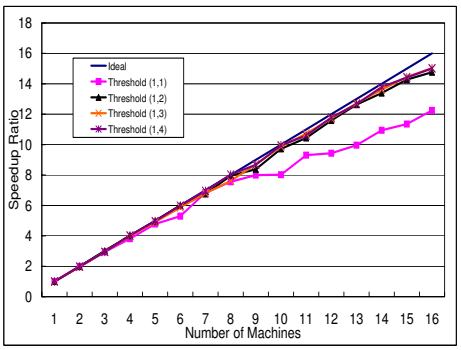

Fig.6 Speedup with MTS-Based Dynamic Load Balancing (Kringle)

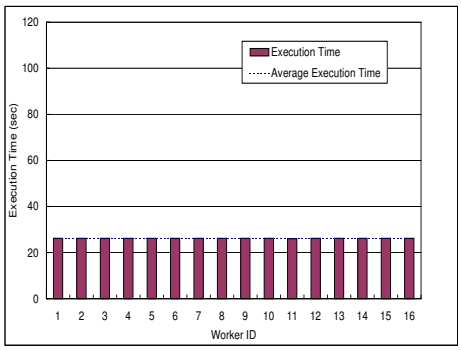

Fig.8 Execution Time of Each Worker Process with MTS-Based Dynamic Load Balancing (Kringle)

Fig.10 and **Fig.11** show the speedup ratios using the dataset including Zinc Finger. In Fig.11, the speedup ratio was greater than the number of machines. This is because processing was slower when using one machine due to a memory shortage. Effective speedup can also be obtained, as it was in the case of using Kringle.

6. Conclusions

This paper presents the MTS-based dynamic load balancing for the PMPS. The speedup of the PMPS with the MTS-based dynamic load balancing is higher than that of a previous implementation. The experimental results show that the MTS-based dynamic load balancing minimizes idle time when distributed workload is unbalanced on the PC clusters.

In the future, it will be necessary to verify the results by using a variety of amino acid sequences.

Acknowledgments This work was supported in part by a Hiroshima City University Grant for Special Academic Research (General Studies, No.3106).

References

1) http://kr.expasy.org/prosite/.
2) R. Agrawal and J. C. Shafer. Parallel mining of association rules. *IEEE Trans. Knowl. Data Eng.*, 8(6):962–969, 1996.

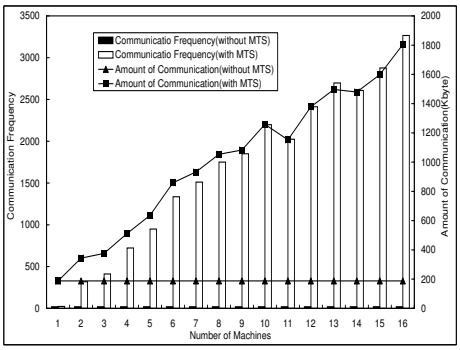

Fig.9 Communication Frequency and Amount of Communication (Kringle)

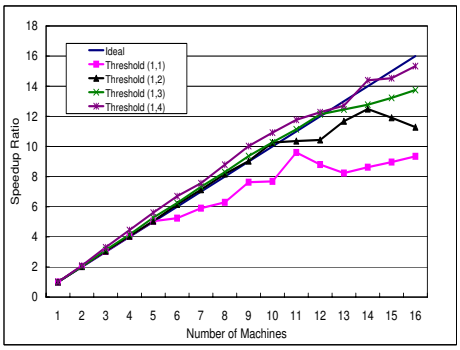

Fig.11 Speedup with MTS-based Dynamic Load Balancing (Zinc Finger)

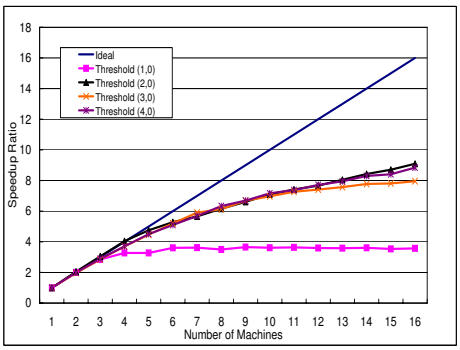

Fig.10 Speedup without MTS-based Dynamic Load Balancing (Zinc Finger)

3) R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 3–14. IEEE Computer Society, 1995.

4) C. Bettini, X. S. Wang, S. Jajodia, and J.-L. Lin. Discovering frequent event patterns with multiple granularities in time sequences. *IEEE Trans. Knowl. Data Eng.*, 10(2):222–237, 1998.

5) J.Han, J.Pei, and Y.Yin. Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2000.

6) H. Kitakami, T. Kanbara, Y. Mori, S. Kuroki, and Y. Yamazaki. Modified prefixspan method for motif discovery in sequence databases. In *PRICAI2002, Proceedings of Trends in Artificial Intelligence, 7th Pacific Rim International Conference on Artificial Intelligence*, volume 2417 of *Lecture Notes in Computer Science*, pages 482–491. Springer, 2002.

7) J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. Prefixspan: Mining sequential patterns by prefix-projected growth. In *Proceedings of the 17th International Conference on Data Engineering*, pages 215–224. IEEE Computer Society, 2001.

8) T. Shintani and M. Kitsuregawa. Parallel mining algorithms for generalized association rules with classification hierarchy. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data*, pages 25–36. ACM Press, 1998.

9) R. Srikant and R. Agrawal. Mining generalized association rules. In *Proceedings of 21th International Conference on Very Large Data Bases*, pages 407–419. Morgan Kaufmann, 1995.

10) T.Sutou, K.Tamura, Y.Mori, and H.Kitakami. Design and implementation of parallel modified prefixspan method. In *High Performance Computing, Proceedings of 5th International Symposium,ISHPC*, volume 2858 of *Lecture Notes in Computer Science*, pages 412–422. Springer, 2003.

11) M.Tamura and M.Kitsuregawa. Dynamic load balancing for parallel association rule mining on heterogenous pc cluster systems. In *Proceedings of 25th International Conference on Very Large Data Bases*, pages 162–173. Morgan Kaufmann, 1999.