Parallel Mining of Closed Sequential Patterns

Shengnan Cong Jiawei Han David Padua
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{cong, hanj, padua}@cs.uiuc.edu

ABSTRACT

Discovery of sequential patterns is an essential data mining task with broad applications. Among several variations of sequential patterns, *closed* sequential pattern is the most useful one since it retains all the information of the *complete pattern set* but is often much more compact than it. Unfortunately, there is no parallel closed sequential pattern mining method proposed yet. In this paper we develop an algorithm, called *Par-CSP* (Parallel Closed Sequential Pattern mining), to conduct parallel mining of closed sequential patterns on a distributed memory system. Par-CSP partitions the work among the processors by exploiting the divide-and-conquer property so that the overhead of interprocessor communication is minimized. Par-CSP applies dynamic scheduling to avoid processor idling. Moreover, it employs a technique, called *selective sampling*, to address the load imbalance problem. We implement Par-CSP using MPI on a 64-node Linux cluster. Our experimental results show that Par-CSP attains good parallelization efficiencies on various input datasets.

Categories and Subject Descriptors: H.2.8 [Database Management]: Database applications–*data mining*; D.1 [Programming Techniques]: Concurrent programming–*parallel programming*

General Terms: Algorithms, Experimentation, Performance

Keywords: parallel algorithms, load balancing, sampling

1. INTRODUCTION

The objective of sequential pattern mining is to discover frequent subsequences in a dataset [1]. Sequential pattern mining has numerous applications, including the discovery of motifs in DNA sequences, the analysis of web log and customer shopping sequences, the study of XML query access patterns, and the investigation of scientific or medical processes. Many efficient sequential pattern mining algorithms have been proposed in the literature [1, 8, 2, 5, 7, 12].

Since a long sequence contains a combinatorial number of subsequences, sequential pattern mining generates an explosive number of frequent subsequences for long patterns, which is prohibitively expensive in both time and space. Therefore, instead of mining the complete set of sequential patterns, an alternative but equally powerful solution is to mine *closed sequential patterns* only. A closed sequential pattern is a sequential pattern which has no supersequence with the same occurrence frequency. Two algorithms have been proposed for mining closed sequential patterns: *CloSpan* [10] and *BIDE* [9]. The former follows a *candidate maintenance-and-test* paradigm over the set of already mined closed sequential pattern candidates. It uses this set to prune the search space and check if a newly found sequential pattern is likely to be closed. Since a large number of closed sequential patterns (or just candidates) would occupy much memory and create a large space for the search of new patterns, using CloSpan for mining long sequences or mining with very low support thresholds tends to be prohibitively expensive. The second algorithm (*BIDE*) adopts a closure checking scheme, called *BI-Directional Extension*, which mines closed sequential patterns without candidate maintenance. Performance studies [9] have shown that BIDE is more efficient than CloSpan.

To make sequential pattern mining practical for large data sets, the mining process must be efficient, scalable, and have a short response time. Moreover, since sequential pattern mining requires iterative scans of the sequence dataset with numerous data comparison and analysis operations, it is computationally intensive. Furthermore, many applications are time-critical and involve huge volumes of data. Such applications demand more mining power than serial algorithms can provide. Thus, it is clearly important to study parallel sequential-pattern mining algorithms that take advantage of the computation and I/O power of distributed memory systems as well as their aggregate memory spaces.

Although a significant amount of research results have been reported on serial implementations of sequential pattern mining, there is still much room for improvement in its parallel implementation. Previous work on parallel sequential-pattern mining has focused on mining the complete set of sequential patterns [11, 3] and, to the best of our knowledge, there is no parallel algorithm that targets closed sequential-pattern mining. Since targeting closed sequential patterns is often more efficient, we decided to follow this approach in the study reported here.

In this paper we develop an algorithm, called *Par-CSP* (Parallel Closed Sequential Pattern mining), to conduct parallel mining for closed sequential patterns on a distributed memory system. Par-CSP:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
KDD'05, August 21–24, 2005, Chicago, Illinois, USA.
Copyright 2005 ACM 1-59593-135-X/05/0008 ...$5.00.

1. is the first parallel algorithm to mine closed sequential patterns;
2. is based on the most efficient serial algorithm *BIDE* to mine the closed sequential patterns without candidate maintenance;
3. is designed for execution on a distributed memory system. Our implementation, achieves good speedups on various datasets.
4. partitions the work into independent tasks so that the overhead of interprocessor communication is minimized;
5. uses *dynamic scheduling* to reduce processor idle time.
6. uses a technique called *selective sampling* for load balancing. Selective sampling accurately predicts the relative times of the subtasks and in this way enables an even distribution of work across processors;

The remainder of the paper is organized as follows. In Section 2, we present a few basic concepts and the serial algorithm on which Par-CSP is based. In Section 3, we describe Par-CSP in detail. The experimental results are presented in Section 4 and in section 5 we discuss related work. Section 6 presents our conclusions.

2. BACKGROUND

2.1 Problem Definition

Let $I = \{i_1, i_2, ..., i_n\}$ be a set of items. A **sequence** s is a tuple, denoted as $\langle T_1, T_2, ..., T_l \rangle$, where $T_j (1 \leq j \leq l)$, called **events** (or itemsets). Each event is a set denoted as $(x_1, x_2, ..., x_m)$ where $x_k (1 \leq k \leq m)$ is an item. For brevity, the brackets are omitted if an element has only one item. A **sequence dataset** S is a set of sequences. The total number of items in a sequence is called the length of the sequence and a sequence with length l is called an l-sequence. A sequence $\alpha = \langle a_1, a_2 ... a_n \rangle$ is called a **subsequence** of another sequence $\beta = \langle b_1, b_2 ... b_m \rangle$, denoted as $\alpha \sqsubseteq \beta$, if there exist integers $1 \leq j_1 \leq j_2 \leq ... \leq j_n \leq m$ such that $a_1 \subseteq b_{j_1}$, $a_2 \subseteq b_{j_2}$,...,$a_n \subseteq b_{j_n}$. If α is a subsequence of β, we say that β contains α. The **support** of a sequence α in a sequence dataset S, denoted $support(\alpha)$, is the number of sequences in the dataset containing α. Given a minimum support threshold, *min_sup*, the set of **sequential pattern**, *SP*, is the set of all the subsequences whose support values are no less than *min_sup*. The set of **closed sequential patterns**, CSP is defined as $CSP = \{\alpha | \alpha \in SP \text{ and } \nexists \beta \in SP \text{ such that } \alpha \sqsubseteq \beta \text{ and } support(\alpha) = support(\beta)\}$. The problem of **closed sequential pattern mining** is to find CSP with support value no less than a minimum support threshold.

2.2 Sequential Algorithm - BIDE

We use *BIDE* [9] as the base serial algorithm for our parallel closed sequential-pattern mining algorithm. We choose BIDE for two reasons. First, BIDE is the most efficient serial algorithm available today to mine closed sequential-patterns. Second, BIDE searches the space without maintaining a set of candidates, which facilitates its parallelization.

The BIDE algorithm only mines sequence dataset consisting of events containing a single item and our parallel algorithm, derived from BIDE, targets the same type of datasets. Extensions have been proposed [9] to make BIDE capable of mining patterns with subsets of items. These extensions can be directly applied to our algorithm because they do not affect the parallel framework proposed in this paper. Focusing on single-item events simplifies our presentation and enables us to focus on the methodology.

Next, we present a brief description of the BIDE algorithm. Let DB be a sequence dataset. The algorithm starts with a scan of DB to identify the frequent 1-sequences. Then, a second scan of DB constructs the *projected datasets* for the frequent 1-sequences. Let i be a sequence, a projection i of DB, denoted as $P(i, DB)$, is a set of subsequences, which are made up of the sequences in DB containing i after deleting the events appearing before the first occurrences of i within each sequence.

For instance, Figure 1 shows a simple sequence dataset. With the support threshold as 2, the projected dataset for sequence AB is $\{C, CB, C, BCA\}$.

Sequence_id	Sequence
10	*C A A B C*
20	*A B C B*
30	*C A B C*
40	*A B B C A*

Figure 1: An example dataset for BIDE

After the projected datasets are built, BIDE searches each projected dataset and enumerates the sequential-patterns following a pattern-growth strategy [4]. Upon getting a sequential-pattern, BIDE applies a closure checking scheme, called *BI-Directional Extension* [9], to check whether the sequential pattern is closed.

If S is a sequence and i is a 1-sequence, $i \diamond S$ represents the concatenation of i and S. Let $\{X_1, X_2, ..., X_n\}$ be a set of sequences, then: $i \diamond \{X_1, X_2, ..., X_n\} \equiv \{i \diamond X_1, i \diamond X_2, ..., i \diamond X_n\}$.

The mining of DB with BIDE can be defined as function $F()$ below where $freq(DB)$ represents the frequent 1-sequences in DB. Function $Check(S)$ returns the sequences in S which can pass the BI-Directional Extension closure checking (closed patterns). We do not describe this check here for lack of space. The reader can find this check in [9]. The closed sequential patterns are stored in set C.

function $F(DB)$
begin
  if (DB is a set of empty sets) **return** NULL;
  else {
    $S = \bigcup_{i \in freq(DB)} ((i \diamond F(P(i, DB))) \cup \{i\})$
    $C = C \cup Check(S)$;
    return(S)
    }
end

3. THE PAR-CSP ALGORITHM

In this section, we introduce an algorithm called Par-CSP to mine the closed sequential-patterns in parallel. We address the following questions: How to decompose BIDE into tasks? How to schedule the resulting tasks? How to balance the load?

3.1 Task Decomposition

BIDE follows three steps:

- Step 1: Identify the frequent 1-sequences

- Step 2: Project the dataset along each frequent 1-sequence;
- Step 3: Mine each resulting projected dataset.

The projected datasets of the frequent 1-sequences are independent. Given a 1-sequence, say i, only the suffixes that follow the first occurrences of i in each sequence are the projection of the dataset along i. Therefore, the closed sequential-patterns mined from the dataset projection along i_1 all start with i_1 as the prefix while the patterns discovered from i_2's projections all start with i_2.

A partition strategy like the one just described is convenient for task decomposition. Since the projected datasets are independent, they can be assigned to different processors. Then, each processor can mine the assigned projected datasets independently by using the conventional BIDE algorithm. No inter-processor communication is needed during the local mining. Our strategy for the parallel mining of closed sequential-patterns is as follows:

1. Each processor counts the occurrence of 1-sequences in a different part of the dataset. A global *add* reduction is executed to obtain the overall counts. The frequent 1-sequences, those that occur at least *min_sup* (the support threshold) times, are identified.

2. For each frequent 1-sequence a very compact representation of the dataset projections, called pseudo-projections, is built. This is done in parallel by assigning a different part of the dataset to each processor. The pseudo-projections are communicated to all processors via an all-to-all broadcast.

3. Dynamic scheduling to distribute the processing of the projections across processors.

To facilitate dynamic scheduling, we assume that the complete dataset is accessible to all processors. In the second step, each processor applies the pseudo-projection method [9] to construct the projected datasets. A pseudo-projection consists of a set of pointers to the starting positions within the dataset of each sequence conforming the projected dataset. After constructing the pseudo-projections, they are broadcast to all processors. In our implementation, we found that it is more efficient to carry out the broadcast using a virtual ring structure where processor I only receives the package from Processor $((I-1)\ mod\ N)$ and only sends the package to Processor $((I+1)\ mod\ N)$. Thus, assume there are total N processors, the all-to-all broadcast is carried out in $(N-1)$ send-receive steps which collectively consume no more than 0.5% of the mining time.

3.2 Task Scheduling

Next, we discuss the mechanism that we use to assign projections to processors.

To reduce load imbalance, Par-CSP uses dynamic scheduling. In our implementation, there is a master processor which maintains a queue of pseudo-projection identifiers. Each of the other processors is initially assigned a projection. After a processor completes the mining of a projection, it sends a request to the master processor for another projection. The master processor replies with the index of the next projection in the queue and removes it from the queue. This process continues until the queue of projections is empty. The requests and replies to and from the master processor are short messages and, therefore, the communication time is usually negligible relative to the mining time.

Dynamic scheduling is quite effective when the subtasks are of similar size and are numerous. However, in many cases, dynamic scheduling cannot achieve load balancing. For example, if the largest subtask takes 25% of the total mining time, the best possible speedup is only 4 regardless of the number of processors available.

For the datasets we used in our experiments, the cost of mining the projections may vary greatly. Figure 2 shows the average and maximum mining time of the projected datasets along frequent 1-sequences for all the datasets we tested (described in Section 4). The relatively large mining time of some projected datasets may result in extremely imbalanced workload. Our experiments prove that the scalability of the parallelization can be greatly improved if the largest projected datasets are partitioned into smaller ones.

	C100S100N5 (sup=0.01%)	C100S50N10 (sup=0.01%)	C200S25N9 (sup=0.01%)	Gazelle (sup=0.2%)
Average	0.387	0.052	1.726	0.102
Maximum	8.442	4.259	349.643	11.663

Figure 2: Mining time distributions

3.3 Relative Mining Time Estimation

Our approach to improving the effectiveness of dynamic scheduling is to identify which projections require long mining time and to further decompose them. To this end, we need to estimate the relative mining time of the projections.

Our strategy to estimate mining time is to use run-time sampling. By mining a small sample of the original dataset and timing the mining time of the projected databases of the sample, we should be able to identify the projections whose mining time is longer. We evaluate sampling strategies by the accuracy of their estimation and the overhead they introduce.

The most natural sampling strategy is *random sampling* which proceeds by collecting a randomly selected subset of the sequences in the dataset, computes the projections, and uses the mining time of this subset to estimate the mining time of each projection. However, we found that random sampling is not accurate if the overhead, which is determined by the size of the subset, is kept small.

We developed an alternative sampling technique, called *selective sampling*, which has proved to be quite accurate in identifying the projections requiring longer mining times. Instead of randomly selecting a subset of sequences from the dataset, selective sampling potentially uses components of every sequence in the dataset.

Selective sampling first discards all infrequent 1-sequences and then discards the last l frequent 1-sequences of each sequence. The number l is computed by multiplying a given fraction t by the average length of the sequences in the dataset. For example, assume $(A:4), (B:4), (C:4), (D:3), (E:3), (F:3), (G:1)$ are the 1-sequences and their counts in the database. Let the support threshold be 4 and the average length of the sequences in the dataset be 4. Suppose t equals to 75% so that l is 3 $(4*.75)$. Then A, B and C are frequent because their support values are no less than the threshold. Given a sequence as $\langle AABCACDCFDB\rangle$, selective sampling will reduce this sequence to $\langle AABCA\rangle$. The suffix $\langle ...CDCFDB\rangle$ is discarded because it contains the last l frequent 1-sequences of the sequence (D and F do not count because they are infrequent 1-sequences.).

Figure 3 uses dataset $C200S25N9$ (described in Section 4)

to compare the mining times obtained with selective sampling with the mining times of the original dataset. The graph shows the mining time of the projections along the frequent 1-sequences for both the complete data set and the dataset resulting from selective sampling. The left vertical scale represents the values for the whole dataset while the one on the right represents the times resulting after selective sampling. The average sequence length of *C200S25N9* is 16 and we set t to 75% so that l is 12. As we can see that the two curves match each other fairly well so that the projections requiring long mining times after selective sampling are also the projections requiring long mining times for the original dataset. The accuracy of selective sampling for all other datasets we studied was similar.

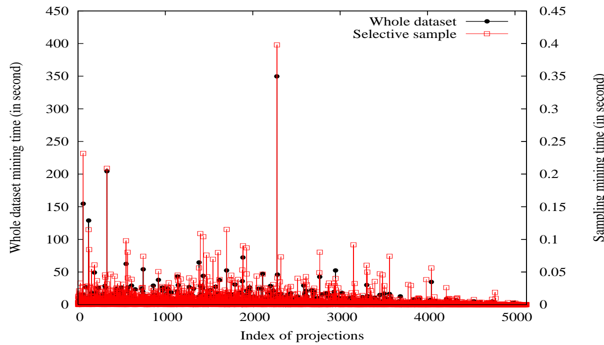

Figure 3: Selective sampling (The $l$arge subtasks in the selective sample are also the $l$arge subtasks in the whole datasets. The estimation is accurate.)

In our implementation, we carry out the mining of the dataset resulting from selective sampling in parallel following the same strategy that we apply to the complete dataset.

As you may expect, there is a trade-off between the accuracy and the overhead of selective sampling. The more frequent 1-sequences we discard, the less accurate selective sampling will be and the less overhead will be introduced by sampling. According to our experiments, 75% is a reasonable value for t for the datasets we considered. For example, with t equal to 75%, the overhead of selective sampling in Figure 3 costs only 0.53% of the serial mining time while it still provides accurate information for the relative mining time estimation. Figure 4 lists the percentage of mining time of selective sampling with t being 75% versus the serial mining time of the whole database.

	C100S100N5	C100S50N10	C200S25N9	Gazelle
Overhead	1.32%	3.97%	0.53%	2.12%

Figure 4: Overhead of selective sampling

Let us now discuss why selective sampling works. When building the projection along a frequent 1-sequence, only the suffixes (with the 1-sequence as prefix) will be collected. The frequent 1-sequences in the tail of a sequence will appear in every projection of their prefixes. Therefore, by removing these frequent 1-sequences, the sequences in most of the projections become shorter and therefore, the mining time can be greatly reduced compared to the mining time of the original dataset. At the same time, the suffixes of the frequent 1-sequences in the tails are shorter so that the mining time of their projections will not be time consuming and, thus, we can safely remove these 1-sequences without significantly affecting the relative times.

3.4 The *Par-CSP* Algorithm

In this subsection, we describe Par-CSP, the parallel algorithm to mine closed sequential-patterns.

Algorithm 1 is the Par-CSP algorithm which is presented in SPMD form. In the first important operation (line 1) each processor counts the 1-sequences for the part of the dataset assigned to it. We assume that the database is partitioned into N subsets and that the subset assigned to processor I is denoted DB_I. In (line 2) an all-to-all reduction is performed to compute the global counts (stored in variable $GLOBAL_COUNTS$ in each processor) and the frequent 1-sequences are identified and stored into variable $F1$. Next (line 3), each processor builds pseudo-projections for the frequent 1-sequences within the assigned portion of the dataset. The pseudo-projections are broadcast to all the processors (line 4). Before scheduling the projections, Par-CSP applies $selective_sampling$ to estimate the relative mining time of these projections (line 5).

Algorithm 1 Par-CSP(I, DB_I, min_sup, CSP_I)

Input: I is the processor ID, DB_I is a portion of the dataset assigned to processor I, min_sup is the minimum support threshold

Output: CSP_I is a portion of closed sequential-patterns

1: $C_I = number_of_1\text{-}sequences(DB_I)$;
2: $GLOBAL_COUNTS = all_to_all_sum(C_I); F1 = frequent_1\text{-}sequences(GLOBAL_COUNTS)$;
3: $PSP = pseudo_projection(F1, DB_I)$;
4: $GLOBAL_PSP = all_to_all_broadcast(PSP)$;
5: $S_RESULT = selective_sampling(F1, I, DB_I, min_sup)$;
6: $F2 = partition(F1, S_RESULT)$; // Partition the most time consuming projections and assign the new set of projections to $F2$.
7: **if** $(I == 0)$ **then**
8: accept requests from slave nodes and reply to each request with a different identifier from set $F2$ until all projections have been assigned;
9: **else**
10: send request for a projection identifier to the master node;
11: stop if all projections have been assigned;
12: apply BIDE algorithm to element of $GLOBAL_PSP$ assigned by the master processor;
13: accumulate the closed sequential-patterns into CSP_I and go back to send request operation;
14: **end if**

The function $selective_sampling()$ implements the process of mining the selective sample which is analog to the process of mining the whole database. But instead of producing the closed sequential patterns, it records the mining time for the projections of all frequent 1-sequences. Variable S_RESULT is assigned these relative mining times.

After the sampling, the top time-consuming projections are partitioned into smaller ones (line 6). In our experiments, we selected those projections whose mining time is more than 3% of the total mining time. For example, if the projection along A is one of the top time consuming projections, it will be partitioned into the projections along

AB, AC and so on. Then the master node schedules these projections as subtasks by maintaining a task queue (line 8). The projections estimated to take longer time in sampling are to be scheduled earlier. Those very small projections can be scheduled in chunks to avoid communication contention. Processor 0 is treated as the master node and taking charge of the task scheduling while all the other processors (slave processors) mine the assigned projections independently without communication to each other. Whenever a slave processor finishes the assigned subtask, it sends a request to the master node for another one until the task queue is empty (line 10-13). Each slave processor outputs the closed sequential patterns in a file. The total closed sequential patterns are simply the concatenation of these files.

4. EXPERIMENTAL RESULTS

4.1 Experimental Setup

Our performance study includes both synthetic and real datasets. We used three synthetic datasets generated by the IBM dataset generator [6] and a real dataset, *Gazelle*, which comes from click-stream data provided by Blue Martini company. In Gazelle, we consider different products as different items and the page views as events. We treat 10 consecutive Web click-stream as a sequence from one customer[1]. The characteristics of these datasets are shown in Figure 5.

Dataset	#seq.	#items	Ave.seq.len .	Max.seq.len .
C100S50N10	100,000	6,044	31	56
C100S100N5	100,000	4,162	62	101
C200S25N9	178,742	5,661	16	39
Gazelle	2,937	1,423	29	1,443

Figure 5: Datasets for experiments

All of our experiments were performed on a Linux cluster consisting of 64 nodes. Each node has a 1GHz Pentium III processor and 1GB main memory. We used MPICH-GM 1.2.4..8a in the implementation of our parallel algorithm. MPICH-GM is a portable implementation of MPI that runs over Myrinet. The operating system is Redhat Linux 7.2 and we used the GNU g++ 2.96 compiler.

4.2 Experimental Results

We first examine the parallel performance of the Par-CSP algorithm. Figure 6 shows the total execution time and the speedup for each dataset. Execution time is measured in seconds throughout this paper. We ran the sequential BIDE algorithm[2] (seq in the figures) and Par-CSP on 4, 8, 16, 32 and 64 processors. As the charts indicate, Par-CSP achieves fairly good performance for all the tested datasets. Par-CSP substantially reduces the mining time comparing to the sequential algorithm. Datasets $C100S100N5$ and $C200S25N9$ whose sequential mining times are larger achieve better speedups than $C100S50N10$ and *Gazelle*.

Another factor that limits the speedups in *Gazelle* is load imbalance. The mining time of some tasks are so large that the subtasks derived from them are still much bigger than the small tasks. The solution to this problem is to apply multi-level task partition. The selective sampling technique can be extended to accomplish this multi-level partitioning. In addition to just recording the mining time corresponding to the frequent 1-sequences during sampling, selective sampling could also record the mining time of their subtasks to identify those which need to be further partitioned.

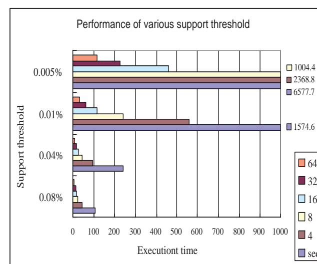

Figure 7: Influence of changing minimum support

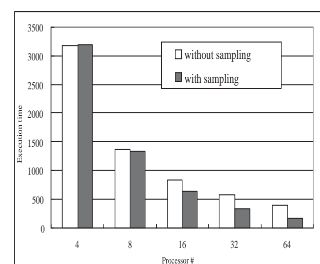

Figure 8: Effectiveness of selective sampling

Next, we discuss the influence of changing minimum support threshold on the performance of Par-CSP. The results are shown in Figure 7. In the figure, we use the dataset $C100S100N5$ with the minimum support threshold varying from a high of 0.08% to a low of 0.005%. We tested Par-CSP on 4, 8, 16, 32 and 64 processors and compared the performance with sequential BIDE algorithm. Par-CSP shows stable parallel performance with different support threshold. Similar results can be obtained for the other datasets.

To test the effectiveness of the selective sampling technique, we compared the performance of Par-CSP when selective sampling is enabled with the performance when it is disabled (Figure 8). We used the dataset $C200S25N9$ with 0.01% as the support threshold. When the number of processors is small, the sampling technique does not show much advantage. This is because when there are only a few processors, the number of subtasks assigned to each processor is large enough so that it tends to balance the load. However, when the number of processors grows larger, the sampling technique can greatly improve the performance. On 64 processors, the performance can be improved by more than 50%.

Previous studies [10, 9] have shown that a serial closed sequential-pattern mining (CSP) algorithm may outperform the serial algorithms for mining all sequential-patterns (ASP) by over one order of magnitude. We performed experiments to compare the parallel performance of mining CSP with the mining of ASP. *PrefixSpan* [7] has been proved to be one of the most efficient sequential algorithms to mine ASP. We implemented an algorithm, Par-ASP, based on *PrefixSpan*, to mine ASP in parallel. We compare the parallel performance of Par-ASP to that of Par-CSP. Here we only show the experimental results for the dataset $C100S100N5$ with the support threshold as 0.01% and 0.005% due to space limitations (Figure 9). The results for the other datasets are similar. In Figures 9(a)(b) we use 0.01% and 0.005% as support threshold respectively. Par-CSP demonstrates a steadily better performance than Par-ASP. It proves that CSPs not only represent more compact results than ASPs but also can lead to better efficiency.

5. RELATED WORK

Although there have been numerous studies on sequential-

[1]We made this choice because the average length of one web click stream is only 3, which makes the serial mining time too short to take advantage of parallelism.

[2]The implementation of BIDE was provided by the algorithm inventor [9].

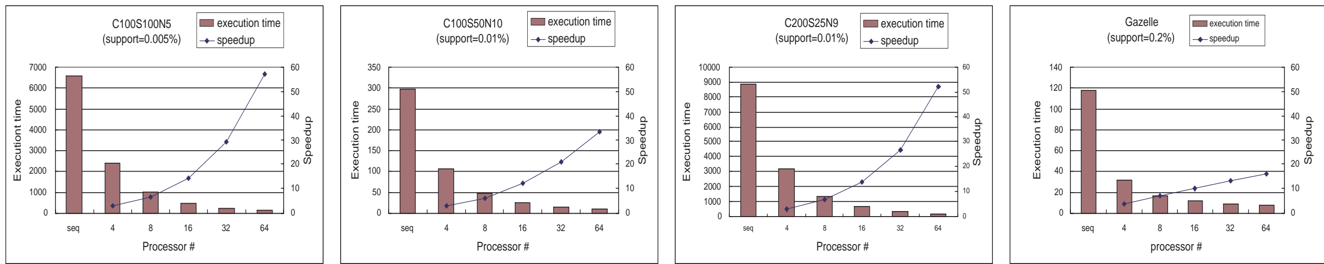

Figure 6: Execution time and speedups of Par-CSP

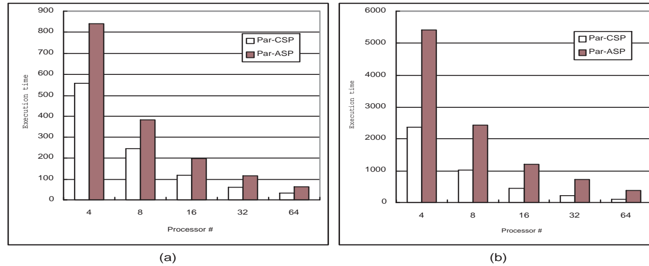

Figure 9: Comparison of Par-CSP with Par-ASP

pattern mining, the study on parallel sequential-pattern mining is still limited and is only confined to mining the complete set of sequential patterns.

In [11], Zaki presents a parallel sequential-pattern mining algorithm, called pSPADE, for discovering the set of *all* frequent subsequences. Different from the Par-CSP algorithm proposed in this paper, pSPADE is targeting a shared-memory system. In a shared memory system, all the processors can access the same global memory space, which makes the proposed recursive dynamic load balancing strategy easy to be implemented. However, applying such a strategy in a distributed memory system, which is typical in a computer cluster environment, is too expensive to be practical. Recently, Guralnik and Karypis [3] presented some parallel sequential-pattern mining algorithms toward a distributed-memory system for mining *the complete set* of sequential-patterns. These parallel algorithms are based on a tree-projection-based sequential algorithm, which is intrinsically similar to the PrefixSpan algorithm [7]. To attack the load balancing problem, the authors proposed a dynamic load-balancing strategy which allows an idle processor to join the busy ones. This strategy involves much more inter-processor communication than our selective sampling approach and the interruption of the busy processors may cause more overhead during mining.

Both of these two parallel formulations still retain the computation efficiency of the underlying serial algorithm to mine the complete set of sequential-patterns. However, mining the complete set of sequential-patterns is usually less efficient than mining the closed sequential-patterns, especially in mining long patterns and with low support threshold, when parallel processing is in greater demand.

6. CONCLUSIONS

In this paper, we propose a parallel closed sequential-pattern mining algorithm *Par-CSP*. It is the first parallel solution for the closed pattern mining problem. We exploit the divide-and-conquer property to minimize the inter-processor communications. We apply dynamic scheduling for task assignment. Furthermore, we devise a technique, called *selective sampling*, to estimate the relative mining time of the subtasks and to achieve load balancing. Our experimental results show that Par-CSP attains good parallelization efficiencies on various input datasets.

7. ACKNOWLEDGMENTS

We are grateful to Dr. Jianyong Wang for providing us the source code of BIDE.

This material is based upon work supported by the National Science Foundation under Grant NGS-0103610 and IIS-0209199. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

8. REFERENCES

[1] R. Agrawal and R. Srikant. Mining sequential patterns. In *Eleventh International Conference on Data Engineering*, pages 3–14, Taipei, Taiwan, 1995.

[2] M. N. Garofalakis, R. Rastogi, and K. Shim. SPIRIT: Sequential pattern mining with regular expression constraints. In *The VLDB Journal*, pages 223–234, 1999.

[3] V. Guralnik and G. Karypis. Parallel tree-projection-based sequence mining algorithms. *Parallel Comput.*, 30(4):443–472, 2004.

[4] J. Han and J. Pei. Mining frequent patterns by pattern-growth: methodology and implications. *SIGKDD Explor. Newsl.*, 2(2):14–20, 2000.

[5] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu. Freespan: frequent pattern-projected sequential pattern mining. In *KDD'00*, pages 355–359. ACM Press.

[6] IBM datset generator forsequential patterns. *http://www.almaden.ibm.com/software/quest/Resources*.

[7] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan mining sequential patterns efficiently by prefix projected pattern growth. In *ICDE'01*, pages 215–226.

[8] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In P. M. G. Apers, M. Bouzeghoub, and G. Gardarin, editors, *Proc. 5th Int. Conf. Extending Database Technology, EDBT*, volume 1057, pages 3–17. Springer-Verlag.

[9] J. Wang and J. Han. BIDE efficient mining of frequent closed sequences. In *ICDE'04*, pages 79–91.

[10] X. Yan, J. Han, and R. Afshar. Clospan: Mining closed sequential patterns in large datasets. In *SDM'03*, 2003.

[11] M. J. Zaki. Parallel sequence mining on shared-memory machines. *Journal of Parallel and Distributed Computing*, 61(3):401–426, 2001.

[12] M. J. Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine Learning*, 42(1-2):31–60, 2001.