

Parallelizing Frequent Web Access Pattern Mining with Partial Enumeration for High Speedup

Peiyi Tang*

Department of Computer Science
University of Arkansas at Little Rock
Little Rock, AR 72204

Markus P. Turkia

Department of Computer Science
University of Arkansas at Little Rock
Little Rock, AR 72204

Abstract

Abstract

The maximum speedup of direct parallelization of pattern-growth mining algorithms for long sequences is limited by the load imbalance among the parallel tasks. In this paper, we present a scheme to parallelize pattern-growth mining algorithms using partial enumeration for high speedup. The experimental results show that partial enumeration increases the achievable speedup of parallel mining significantly for the databases with long sequences.

1 Introduction

A sequence is the fundamental mechanism to encode information and design as demonstrated by the biological sequences. This is the reason why data mining of frequent patterns from sequences including web access sequences has attracted significant attention in recent years [1, 2, 3, 4, 5].

The complexity of mining frequent patterns from sequence databases is inherently exponential due to the fact that patterns are sequences themselves. The search space of frequent patterns grows exponentially as the length of the patterns increases. One way to cope with this complexity is to use parallel computers for the mining. In this paper, we focus on parallelizing the pattern-growth frequent pattern mining algorithms [1, 5, 6].

The major problems in direct parallelizing the pattern-growth algorithms are two-fold: (1) Because the pattern-growth algorithms are recursive algorithms, the number of parallel tasks that can be found is limited by the number of the symbols used in the sequences. (2) According to empirical studies, the execution times of these parallel tasks vary significantly.

The maximum speedup achievable is limited by the longest task, even though more parallel processors are available.

In this paper, we propose to use *partial enumeration* to parallelize the pattern-growth mining algorithms to solve the problems above. Partial enumeration is a technique to extend the pattern-growth algorithms so that the frequent patterns can grow faster with more than one symbol at a time [7]. It was shown in [7] that the sequential mining time can be reduced by partial enumeration for the databases with long sequences. The purpose of using partial enumeration in this paper is to increase the number of tasks beyond the number of symbols and, more importantly, to have better load-balanced parallel tasks to achieve high speedup. Our experimental results show that partial enumeration can increase the speedup of parallel mining significantly for the databases with long sequences.

The rest of the paper is organized as follows. In Section 2, we introduce the technique of partial enumeration and show why it can increase the maximum speedup achievable in the parallel mining. In Section 3, we describe the scheme to parallelize the pattern-growth mining algorithm [6] using partial enumeration [7]. The experimental results and the conclusion of the paper are presented in Section 4.

2 Partial Enumeration for High Speedup

Let Σ be the set of symbols used to construct sequences. A *web access sequence* s is a sequence of finite number of symbols from Σ , $s = \sigma_1 \cdots \sigma_m$ ($\sigma_i \in \Sigma$ for all $1 \leq i \leq m < \infty$ and s_i and s_j are not necessarily different for $i \neq j$). A *web access database* D is a multi-set of web access sequences. A *pattern* is also a web access sequence. A web access sequence $s' = \sigma'_1 \cdots \sigma'_n$ is a *subsequence* of sequence $s = \sigma_1 \cdots \sigma_m$, denoted as $s' \subseteq s$, if and only if there exist i_1, \dots, i_n ($n \leq m$) such

*Supported in part by NASA and the Arkansas Space Grant Consortium under Grant UALR11604.

that $1 \leq i_1 < \dots < i_n \leq m$ and $\sigma'_j = \sigma_{i_j}$ for all $1 \leq j \leq n$. A web sequence s is said to *support* a pattern p if p is a subsequence of s . The *support* of pattern p in database D , denoted as $Sup_D(p)$, is the number of sequences in D that support p . Given a threshold ξ in interval $(0, 1]$, a pattern p is frequent with respect to ξ and D if $Sup_D(p) \geq \xi|D|$, where $|D|$ is the number of sequences in D . $\xi|D|$ is called the *absolute threshold* and denoted by η . The web access pattern mining problem is to find all the frequent patterns with respect to ξ and D .

Figure 1 illustrates the search space for the frequent patterns from the symbol set $\Sigma = \{a, b, c\}$.

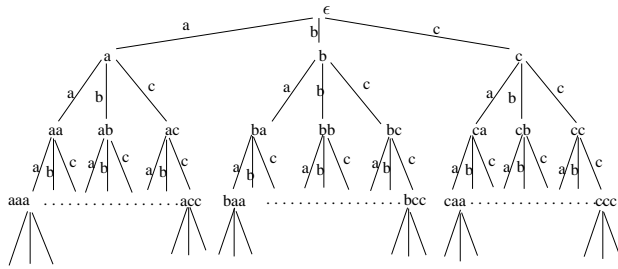


Figure 1: Search Space Tree

The pattern-growth algorithms [1, 5, 6] grow frequent patterns by one symbol at a time. The abstract form of the pattern-growth algorithm [6] is shown in Figure 2. The pattern-growth algorithms [1, 5, 6] are

```

function Pattern-Grow(pattern  $q$ , database  $D$ ) {
   $F \leftarrow \emptyset$ ;
  for each  $\sigma$  in  $\Sigma$  do
    if ( $Sup_D(\sigma) \geq \eta$ ) then
       $F \leftarrow F \cup \{q \cdot \sigma\}$ ;
      Construct projection database  $D_\sigma$ ;
       $F' \leftarrow$  Pattern-Grow( $q \cdot \sigma, D_\sigma$ );
       $F \leftarrow F \cup F'$ ;
    endif
  endfor
  return  $F$ ;
}

```

Figure 2: Pattern-Growth without Enumeration

all based on the principle of conditional searching. D_σ is the σ -projection database of D . It consists of all the σ -projections of the sequences in D that support σ . The σ -projection of a sequence is what left after the prefix from the first symbol up to the first occurrence of σ (inclusive) is deleted. For example, the b -projection of $aab\underline{b}cba$ is $cb\underline{a}$ because its prefix up to the first occurrence of b as underlined is aab . It can be proved that the support of pattern p in D_σ

is equal to the support of pattern $\sigma \cdot p$ in D , i.e. $Sup_D(\sigma \cdot p) = Sup_{D_\sigma}(p)$. The details for the principle of conditional searching can be found in [6]. The set of all frequent patterns in D , thus, can be found by invoking *Pattern-Grow*(ϵ, D) in Figure 2 (ϵ is the empty pattern).

Since the pattern-growth algorithms are all recursive algorithms as shown in Figure 2, the only way to parallelize them is to turn the **for** loop of the top level call into a parallel loop and allocate the tasks of mining the frequent patterns starting with individual symbols to parallel processors. It is obvious that the maximum speedup of this parallelization is bound by the number of symbols in Σ . For example, we can use only three processors for the parallel mining in the search space in Figure 1 and the maximum speedup is 3. Many important applications do not have large number of symbols (DNA sequences have four symbols of nucleotides and protein sequences have 20 symbols of amino acids).

The maximum speedup is further limited by the load imbalance of these tasks. We conducted an empirical study about the execution times of these tasks and found that they vary significantly.

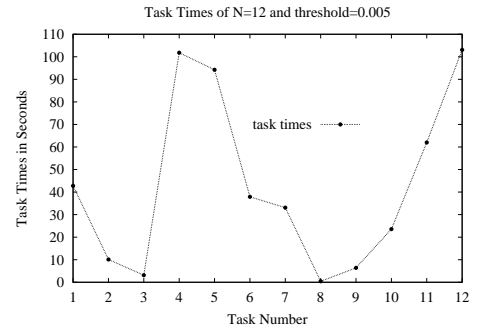


Figure 3: Task Times for N=12, C=16, D=1000

Figure 3 shows the task times of the 12 tasks of the FLWAP-tree mining [6] on a 497 Mhz Pentium III processor for a dataset with $N = 12$ symbols generated by the IBM data generator. The dataset has $D=1000$ sequences and the average length of the sequences is $C = 16$. The threshold ξ is 0.005. The task with the largest execution time, which we called super-task, is task 12. It takes 103.079 seconds to complete. The total execution time of all the tasks is 518.561 seconds. If we run the mining on parallel processors, the parallel execution time is at least the task time of the supertask. The maximum speedup that can be achieved is, thus, $518.561/103.079 = 5.03$, even though more parallel processors are available.

In general, if T_1, \dots, T_n be the task times of the total n parallel tasks of mining, the maximum speedup

achievable by parallel mining is

$$SP_M = \frac{T_1 + \dots + T_n}{\max(T_1, \dots, T_n)} \quad (1)$$

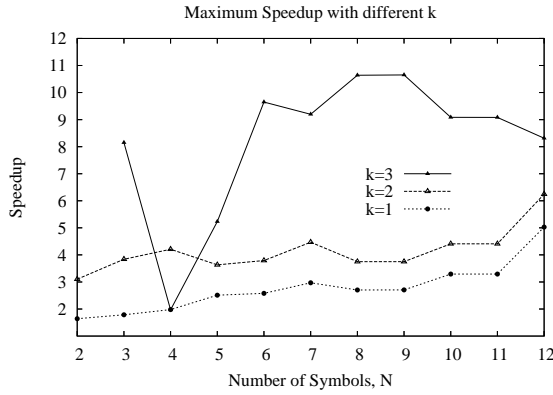


Figure 4: Maximum Speedups SP_M

We also conducted an empirical study of the task times for the datasets with $N = 2, \dots, 12$ symbols, $C = 16$ and $\xi = 0.005$. The maximum speedups for these datasets calculated by (1) are plotted on the line labeled “k=1” in Figure 4. They are between 1.64 and 5.02, way below the number of the tasks (which is N), due to the load imbalance of the tasks.

In order to raise the maximum speedup for parallel mining, the work loads of parallel tasks need to be better balanced. We have extended the pattern-growth mining with partial enumeration [7] to grow the frequent patterns by more than one symbol at a time¹. Figure 5 [7] shows the re-arranged search space

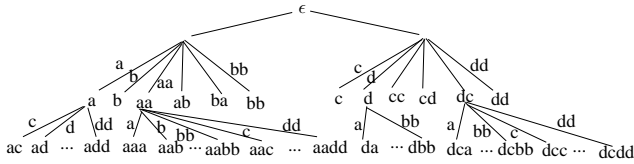


Figure 5: Search Space by Partial Enumeration

to illustrate the idea of partial enumeration. The symbol set $\Sigma = \{a, b, c, d\}$ is first partitioned to disjoint subsets $\Sigma_1 = \{a, b\}$ and $\Sigma_2 = \{c, d\}$. For each subset, the enumerations of the non-empty patterns of length no more than the size of the subset are called *partial enumerations*. For example, the partial enumerations from $\Sigma_1 = \{a, b\}$ are a, b, aa, ab, ba, bb . The set of partial enumerations from the symbol subset Σ_j

¹The experimental evaluation shows that the pattern-growth mining with partial enumeration [7] outperforms the pattern-growth mining without enumeration [6] for the datasets with long sequences.

```

function Pattern-Grow(pattern  $q$ , int  $i$ , database  $D$ ) {
   $F \leftarrow \emptyset$ ;
  for  $j = 1, l$  do
    if ( $j \neq i$ ) then
      Construct apriori enumerator  $Enum(\Sigma_j)$ ;
1:   while ( $p \leftarrow Enum.Next()$ ) is not null) do
2:     if ( $(Sup)_D(p) \geq \eta$ ) then
3:       Construct the  $p$ -projection database  $D_{p, \Sigma_j}$ ;
       Call  $Enum.Confirm(p)$  to report  $p$  is frequent;
        $F \leftarrow F \cup \{q \cdot p\}$ ;
       if ( $|p| = |\Sigma_j|$ ) then
          $F' \leftarrow Pattern-Grow(q \cdot p, 0, D_{p, \Sigma_j})$ ;
       else
          $F' \leftarrow Pattern-Grow(q \cdot p, j, D_{p, \Sigma_j})$ ;
       endif
        $F \leftarrow F \cup F'$ ;
     endif
  endwhile
  Delete apriori enumerator  $Enum$ ;
endif
endfor
return  $F$ ;
}

```

Figure 6: Pattern-Growth with Partial Enumeration

is denoted by $E(\Sigma_j)$. The search space is re-arranged as follows. The empty pattern ϵ at the root node grows with the partial enumerations from both $\Sigma_1 = \{a, b\}$ and $\Sigma_2 = \{c, d\}$. For each of the other nodes, if it was grown from its parent node with a partial enumeration shorter than the size of the symbol subset, it will not be extended with the same symbol subset. The reason is to prevent repeating the same pattern in the search space tree. For example, pattern a is grown from the empty pattern ϵ with partial enumeration a whose length is less than 2, the size of $\Sigma_1 = \{a, b\}$. Therefore, pattern a grows with the partial enumerations from $\Sigma_2 = \{c, d\}$ only, namely c, d, cc, cd, dc, dd . If it grows with the partial enumerations from $\Sigma_1 = \{a, b\}$, we would have patterns aa and ab as its children. But, aa and ab are already enumerated along with a as the children of the root node ϵ . Patterns aa and ab would be mined twice if they are frequent. On the other hand, if a node is grown from its parent with a partial enumeration of the length equal to the size of the symbol subset, it should be extended with the partial enumerations of *all* symbol subsets. For instance, pattern aa is grown from the parent node (the empty pattern ϵ) with partial enumeration aa whose length is 2, the size of $\Sigma_1 = \{a, b\}$. It should continue to grow with the partial enumerations from both $\Sigma_1 = \{a, b\}$ and $\Sigma_2 = \{c, d\}$. In particular, the growth from pattern aa with the partial enumerations from $\Sigma_1 = \{a, b\}$

```

function Mine(enum  $p$ , int  $j$ , database  $D$ ) {
  if ( $Sup_D(p) \geq \eta$ ) then
    Report to the Master that  $p$  of  $E(\Sigma_j)$  is frequent;
     $F \leftarrow \{p\}$ ;
    Construct projection database  $D_{p,\Sigma_j}$ ;
    if ( $|p| = |\Sigma_j|$ ) then
       $F' \leftarrow Pattern-Grow(p, 0, D_{p,\Sigma_j})$ ;
    else
       $F' \leftarrow Pattern-Grow(p, j, D_{p,\Sigma_j})$ ;
    endif
     $F \leftarrow F \cup F'$ ;
    Send  $F$  to the Master and request a next task;
  else
    Report to Master that  $p$  of  $E(\Sigma_j)$  is
    infrequent and request a next task;
  endif
}

main-slave() {
  Input the database  $D$ ;
  Send a task Request to Master Processor;
  Receive a task  $(p, j)$  from Master Processor;
  while (task is not null) do
    Mine( $p, j, D$ );
    Receive a task  $(p, j)$  from Master Processor;
  endwhile
}

```

Figure 7: Algorithm for Slave Processors

will not duplicate any patterns enumerated.

When enumerating the partial enumerations from $E(\Sigma_j)$, the downward enclosure property should be used to prune away the infrequent enumerations if their subsequences are known to be infrequent. For example, if a from $E(\{a, b\})$ is infrequent, then aa, ab, ba from $E(\{a, b\})$ are also infrequent and should not be enumerated. This is called *apriori partial enumeration*. Figure 6 shows the frequent pattern mining algorithm with apriori partial enumeration [7]. The apriori partial enumerations p from $E(\Sigma_j)$ are obtained by calling the $Next()$ function of the apriori partial enumerator $Enum$, created for Σ_j . If p is frequent in D ($Sup_D(p) \geq \eta$), the p -projection database of D , denoted as D_{p,Σ_j} , is constructed. The apriori partial enumerator $Enum$ is also informed that p is frequent. D_{p,Σ_j} consists of p -projections of the sequences in D that support p . The p -projection of sequence s supporting p is what left after the p -prefix of s is deleted. The p -prefix of s is the minimal prefix of s that supports p . It is assumed in Figure 6 that the symbol set Σ is partitioned to l subsets, $\Sigma = \Sigma_1 \cup \dots \cup \Sigma_l$. It can be proved that each frequent pattern will be mined only once by the algorithm in Figure 6 [7]. The set of frequent patterns in the original database D is mined

by calling $Pattern-Grow(\epsilon, 0, D)$.

Each apriori partial enumeration p obtained in line 1 of Figure 6 represents a task of mining the corresponding projection database, D_{p,Σ_j} . We conducted an empirical study of the execution times of these tasks and found that the work loads of these tasks are much better balanced. As a result, the maximum speedup calculated by (1) is raised. We partition the symbol set Σ to $l = \lceil \frac{N}{k} \rceil$ subsets, where $N = |\Sigma|$. The first $l - 1$ subsets have the size of k and the size of the last subset may be less than k . When $k = 1$ (and thus $l = N$), the algorithm in Figure 6 degenerates to the mining algorithm without enumeration as shown in Figure 2. The plots labeled “k=2” and “k=3” in Figure 4 are the maximum speedups of the tasks with partial enumeration for the same datasets with $N = 2, \dots, 12$, $C=16$, and $\xi=0.005$. The maximum speedup for $k = 2$ is higher than that of $k = 1$ for every value of N . When $k = 3$, the maximum speedups are higher than $k = 2$ except for the case of $N = 4$. These empirical studies show that partial enumeration can raise the maximum speedup achievable in parallel mining of frequent patterns.

3 Parallelization

The parallel tasks represented by apriori partial enumerations p from $E(\Sigma_j)$ for all j , denoted by $task(p, j)$, are scheduled by the master processor and executed by the slave processors. The code for parallel task (p, j) is shown in Figure 7.

Each slave processor starts by inputting the original database D and sending a task request to the master processor. Then it receives a task (p, j) and executes it by calling $Mine(p, j, D)$ repeatedly until it receives a null task signifying that all the tasks have been allocated. The main function of slave processors is $main-slave()$ shown in Figure 7.

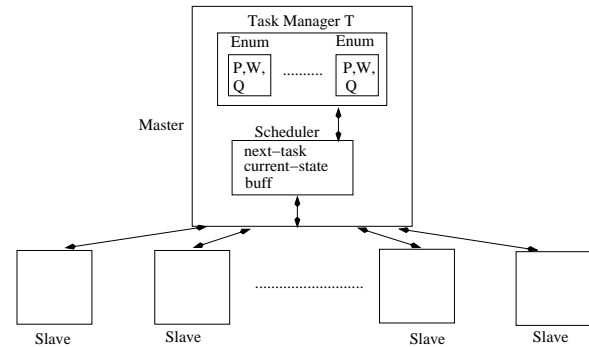


Figure 8: Software Architecture of Parallel Processors

Figure 8 shows the software architecture of the

system. The master processor runs a dynamic scheduler. The basic function of the scheduler is to interact with the task manager T to get parallel tasks and allocate them to the slave processors dynamically. The scheduler of the master processor communicates with the slave processors through the messages as follows:

- Slave processor S_i sends *task request* $Req(S_i)$ to the master processor.
- The master processor sends *task* (p, j) (or a *null task* signifying that all the tasks are allocated) to slave processor S_i in response to the task request $Req(S_i)$.
- Slave processor S_i sends a *Report* to the master processor about whether p is frequent in D in response to the task (p, j) .
- Slave processor S_i sends the set of frequent patterns it mined for the task (p, j) to the master processor, if p is frequent.

```

Upon Receiving task request  $Req(S_i)$  from  $S_i$  {
  if (current-state is wait) then
    Enqueue  $S_i$  into buffer buff;
  elseif (current-state is ready) then
    Send task  $(p, j)$  held in next-task to  $S_i$ ;
     $T.Update(current-task, current-state)$ ;
  else
    // current-state is done
    Send null task to slave processor  $S_i$ ;
  endif
}

Upon Receiving a report  $Report(p, j, freq)$  {
  Pass the report to the task manager  $T$  by
  calling  $T.Report(p, j, freq)$ ;
  if (current-state is wait) then
     $T.Update(current-task, current-state)$ ;
  endif
  while (current-state is ready and buff is not empty) do
     $S \leftarrow buff.Dequeue()$ ;
    Send task  $(p, j)$  held in next-state to slave processor  $S$ ;
     $T.Update(current-task, current-state)$ ;
  endwhile
  while (current-state is done and buff is not empty) do
     $S \leftarrow buff.Dequeue()$ ;
    Send the null task to slave processor  $S$ ;
  endwhile
}

```

Figure 9: The Protocol of the Scheduler

As shown in main-slave() in Figure 7, the slave processors piggy-back the request for the next task to the

report of infrequent p or the set of frequent patterns mined. However, the report that p is frequent is sent as soon as it is known before the mining starts. This allows the master to generate further parallel tasks at the earliest possible time.

The scheduler has two variables *next-task* and *current-state* (see Figure 8). The *next-task* holds the next task (p, j) ready to be dispatched to any slave processor that requests a task. The *current-state* tells the current state of the scheduler. It can hold one of the three states as follows:

- *ready*: *next-task* holds a task ready to be dispatched.
- *wait*: There is no task ready to be dispatched and further tasks are pending.
- *done*: All the tasks have been generated.

The task manager T has two functions to be called by the scheduler:

- $Update(next-task, current-state)$ to update *next-task* and *current-state*.
- $Report(p, j, freq)$ to report that the pattern p of task (p, j) is frequent (*freq* being true) or not (*freq* being false).

The main function of the scheduler is a finite state machine protocol responding to the task requests and the reports sent from the slave processors. The scheduler also has a buffer queue, *buff*, to hold the slave processors that are waiting for tasks. The protocol of the scheduler is shown in Figure 9.

The task manager T manages l apriori enumerators $Enum_j$ ($j = 1, \dots, l$), one for each symbol subset Σ_j . The details of the task manager T can be found in [8].

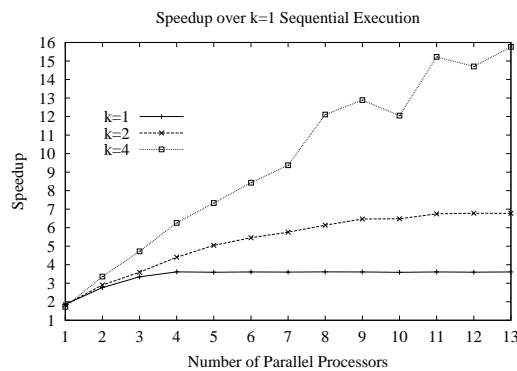
4 Experimental Results and Conclusion

We have implemented our parallel web access pattern mining with MPI on a cluster of workstations. Two datasets, N4C17D1KS4 and N6C16D1KS4, are generated by the IBM data generator for the experiments. In the names of the datasets, we use N for the number of symbols, i.e. $|\Sigma|$, C for the average length of the sequences of the database, D for the size of the database (in thousands). Both datasets were mined using threshold $\xi = 0.005$.

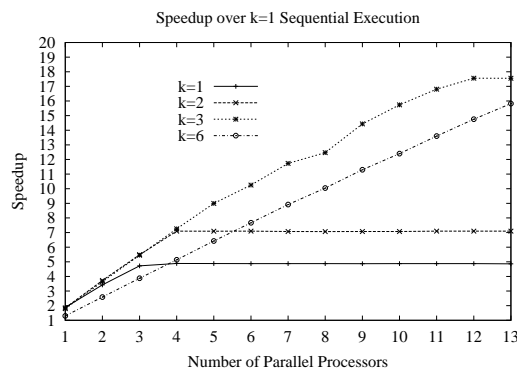
We run the parallel mining for the datasets N4C17D1KS4 and N6C16D1KS4 with $k = 1, 2, 4$ and $k = 1, 2, 3, 6$, respectively. The number of parallel

slave processors varying from 1 to 13. Each slave processor is a 497 Mhz Pentium III processor. The master is a 997 Mhz Pentium III processor.

Figures 10(a) and 10(b) show the speedup of the parallel executions over the sequential mining without enumeration [6] of datasets N4C17D1KS4 and N6C16D1KS4, respectively. Figure 10(a) shows that $k = 4$ allows the speedup to increase as the number of processors increases. The $k = 3$ and $k = 6$ do the same for dataset N6C16D1KS4 as shown in Figure 10(b). This is because with large k values the partial enumeration generates more and better load-balanced parallel tasks and, thus, allows more parallel processors to be used to reduce the execution time. Note that the speedup of $k = 3$ for N6C16D1KS4 is higher than that of $k = 6$. This is because with $k = 6$ the partial enumeration generates many smaller parallel tasks and the communication overhead between the master and slave processors slows down the parallel execution.



(a) Dataset N4C17D1KS4



(b) Dataset N6C16D1KS4

Figure 10: Speedup over Sequential Mining

We have presented a scheme to parallelize the pattern-growth frequent pattern mining algorithms using partial enumeration. We have shown that partial enumeration can increase the speedup achievable by providing better load-balanced parallel tasks.

Previous works on parallelizing the sequential pattern mining such as [9] use direct parallelization and only the databases with short sequences are used.

References

- [1] Jian Pei, Jiawei Han, Behzad Mortazavi-asl, and Hua Zhu. Mining access patterns efficiently from web logs. In *Proceedings of the 4th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'00)*, pages 396–407. Lecture Notes in Computer Science, Vol. 1805, 2000.
- [2] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu. Sequential pattern mining using a bitmap representation. In *Proceedings of the eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 429–435, 2002.
- [3] J. Wang and J. Han. BIDE: Efficient mining of frequent closed sequences. In *Proceedings of the 20th International Conference on Data Engineering (ICDE'04)*, pages 79–90, 2004.
- [4] Ke Wang, Yabo Xu, and Jeffery Xu Yu. Scalable sequential pattern mining for biological sequences. In *Proceedings of the Thirteenth ACM International Conference on Information and knowledge management (CIKM'04)*, pages 178–187, November 2004.
- [5] C.I. Ezeife and Yi Lu. Mining web log sequential patterns with position coded pre-order linked WAP-tree. *International Journal of Data Mining and Knowledge Discovery*, 10:5–38, 2005.
- [6] Peiyi Tang, Markus P. Turkia, and Kyle A. Gallivan. Mining web access patterns with first-occurrence linked WAP-trees. In *Proceedings of the 16th International Conference on Software Engineering and Data Engineering (SEDE'07)*, pages 247–252, Las Vegas, USA, July 2007.
- [7] Peiyi Tang and Markus P. Turkia. Mining frequent web access patterns with partial enumeration. In *Proceedings of the 45th Annual Association for Computing Machinery Southeast Conference (ACMSE'07)*, pages 226–231, Winston-Salem, NC, USA, March 2007.
- [8] Peiyi Tang and Markus P. Turkia. Parallelizing frequent web access pattern mining with partial enumeration for high speedup. Technical Report titus.compsci.ualr.edu/~ptang/papers/parwap.pdf, Dept of CS, UALR, 2007.
- [9] Shengnan Cong, Jiawei Han, and David Padua. Parallel mining of closed sequential patterns. In *Proceeding of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, pages 562–567, 2005.