Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 00-002

Parallel Algorithms for Mining Sequential Associations: Issues and Challenges

Mahesh Joshi, George Karypis, and Vipin Kumar

January 12, 2000

Parallel Algorithms for Mining Sequential Associations: Issues and Challenges *

Mahesh V. Joshi† George Karypis† Vipin Kumar†

Abstract

Discovery of predictive sequential associations among events is becoming increasingly useful and essential in many scientific and commercial domains. Enormous sizes of available datasets and possibly large number of mined associations demand efficient and scalable parallel algorithms. In this paper, we first present a concept of universal sequential associations. Developing parallel algorithms for discovering such associations becomes quite challenging depending on the nature of the input data and the timing constraints imposed on the desired associations. We discuss possible challenging scenarios, and propose four different parallel algorithms that cater to various situations. This paper is written to serve as a comprehensive account of the design issues and challenges involved in parallelizing sequential association discovery algorithms.

*This work was supported by NSF grant ACI-9982274, by Army Research Office grant DA/DAAG55-98-1-0441, by Army High Performance Computing Research Center cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008, the content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred. Access to computing facilities was provided by AHPCRC, Minnesota Supercomputer Institute. Related papers are available via WWW at URL: http://www.cs.umn.edu/~kumar.

†Department of Computer Science, University of Minnesota, Minneapolis, MN 55455

1 Introduction

One of the important problems in data mining [CHY96] is discovering associations present in the data. Such problems arise in the data collected from scientific experiments, or monitoring of physical systems such as telecommunications networks, or from transactions at a supermarket. In general, this data is characterized in terms of objects and events happening on these objects. As an example, a customer can be an object and items bought by him/her can be the events. In experiments from molecular biology, an organism or its chromosome can be an object and its behavior observed under various conditions can form events. In a telecommunication network, switches can be objects and alarms happening on them can be events. The *events* happening in such data are related to each other via the temporal relationships of *together* and *before* (or *after*). The *association rules* originally proposed in [AIS93] utilize only the *together* part of the relationship to come up with associations between the events. The concept was extended to the discovery of *sequential patterns* [AS96] or *episodes* [MTV95], which take into account the sequential (*before/after*) relationship as well. The formulation in [AS96] was motivated by the supermarket transaction data, and the one in [MTV95] was motivated by the telecommunication alarm data. Formulation in [JKK99] unifies and generalizes these formulations.

The algorithms that discover sequential associations are mainly motivated by those developed for non-sequential associations. Various serial and parallel algorithms have been proposed to discover non-sequential association rules (or simply, associations rules). The most time consuming operation in this discovery process is the computation of the frequencies of the occurrence of candidate subsets of events. Many databases have a large number of distinct events, which yields prohibitively large number of candidates. Hence, most current association rule discovery techniques try to prune the search space by requiring a minimum level of support for candidates under consideration. Support is defined as the number of occurrences of the candidates in the database transactions. *Apriori* [AS94] is a one of the first and most widely used algorithms that aggressively prunes the set of potential candidates of size k by using the following observation: a candidate of size k can meet the minimum level of support only if all of its subsets also meet the minimum level of support. Candidates for $(k+1)^{st}$ iteration are generated using only those event-sets from k^{th} iteration that satisfy the support requirement. Efficient techniques, such as hash trees, are used to count the occurrence frequencies of candidates. The parallel algorithms developed for non-sequential associations [HKK97] try to effectively parallelize the phases of candidate counting and candidate generation.

The sequential nature of the data, depicted by the *before/after* relationships, is important from the discovery point of view as it discovers more powerful and predictive associations. It is important from the algorithmic point of view also, as it increases the complexity of the problem enormously. The number of sequential associations possible is much larger than non-sequential associations. Various formulations proposed so far [AS96, MTV95, SA96, JKK99], try to contain the complexity by imposing various temporal constraints, and by using the monotonicity of the support criterion as the number of events in the association increases (the Apriori principle). In this paper, we mainly concentrate on the discovery of general-

ized universal sequential patterns proposed in [JKK99]. It is believed that this formulation is applicable over a wide range of application domains, and encapsulates different ways of defining interestingness of the discovered associations. We present a serial algorithm that discovers these universal sequential associations. The enormity and high dimensionality of the data can make these serial algorithms computationally very expensive; and hence, efficient parallel algorithms are very essential for discovering sequential associations. Different issues and challenges arise mainly due to the sequential nature of the associations, and the way in which their interestingness measure is defined (counting strategies). We discuss all these issues and challenges, and propose formulations for resolving them. Essential idea is to extend the parallel formulations used for non-sequential associations [HKK97], so as to take into account the challenges introduced by the sequential nature. In the first formulation called *Event Distribution* (*EVE*), we distribute input data alone and replicate the candidate hash tree on all the processors. Different parallelization strategies are possible depending on the number of objects in the input data, the number of events happening on these objects, and the timing constraints. Three algorithms *Simple Event Distribution* (*EVE-S*), *Event Distribution with Partial Replication* (*EVE-R*), and *Event Distribution for Complex Scenario* (*EVE-C*) are proposed. EVE formulations might become inefficient when the number of candidate sequences becomes large, because they generate the candidates serially and these candidates are replicated on all processors. In the second formulation called *Event and Candidate Distribution* (*EVECAN*), we partition the input data similar to *EVE* and also partition the candidates. The candidate generation phase is parallelized using a distributed hash table mechanism, whereas the counting phase is parallelized using an approach similar to IDD[HKK97] approach used for non-sequential associations.

The rest of this paper is organized as follows. Section 2 contains a description of a serial algorithm for finding sequential associations. In Section 3, we elaborate on the challenges of parallelizing these algorithms, and propose different parallel formulations. Section 4 contains conclusions.

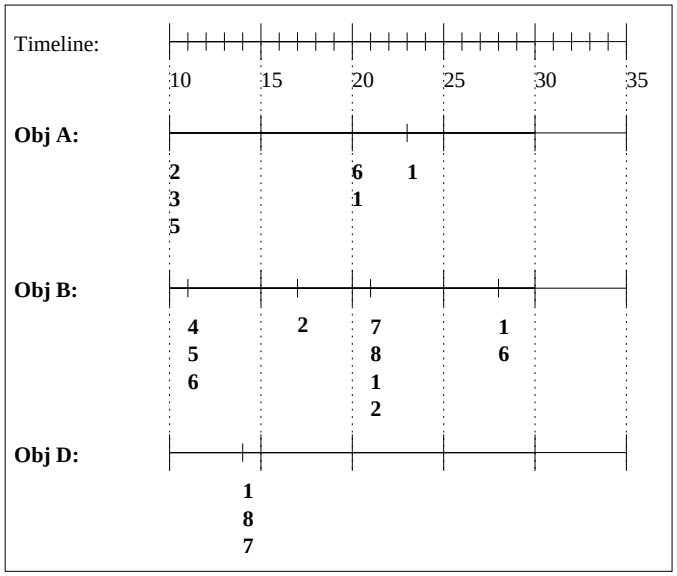
2 Mining the Sequential Relationships

The data collected from scientific experiments, or monitoring of physical systems such as telecommunications networks, or from transactions at a supermarket, have inherent sequential nature to them. Sequential nature means that the events occurring in such data are related to each other by relationships of the form *before* (or *after*) and *together*. This information could be very valuable in finding more interesting patterns hidden in the data, which could be useful for many purposes such as prediction of events or identification of better sequential rules that characterize different parts of the data.

In this section, we discuss the concept of *sequential associations*, more commonly known as *sequential patterns*, and serial algorithms to discover them.

Object	timestamp	events
A	10	2, 3, 5
A	20	6, 1
A	23	1
B	11	4, 5, 6
B	17	2
B	21	7, 8, 1, 2
B	28	1, 6
D	14	1, 8, 7

(a)



(b)

Figure 1: Example Input Data: (a) Flat representation, (b) Timeline Representation

2.1 Generalized Sequential Associations: Definition

Sequential associations are defined in the context of an input sequence data characterized by three columns: *object*, *timestamp*, and *events*. Each row records occurrences of events on an object at a particular time. An example is shown in Figure 1(a). Alternative way to look at the input data is in terms of the time-line representations of all objects as illustrated in Figure 1(b). Note that the term *timestamp* is used here as a generic term to denote a measure of sequential (or temporal) dimension.

Various definitions of *object* and *events* can be used, depending on what kind of information one is looking for. For example, in one formulation, object can be a telecommunication switch, and event can be an alarm type occurring on the switch. With this, the sequences discovered will indicate interesting patterns of occurrences of alarm types occurring at a switch. In another formulation, object can be a *day*, and event can be a switch or a pair of switch and type of the alarm occurring on it. This will give interesting sequential relations between different switches or switch-alarm type pairs over a day.

Given this input data, the goal is to discover associations or patterns of the form given in Figure 2. A pattern is essentially a sequence of sets of events, which conform to the given *timing constraints*. As an example, the sequential pattern (A) (C,B) (D), encodes an *interesting* fact that event D occurs after an event-set (C,B), which in turn occurs after event A. The occurrences of events in a sequential pattern are governed by the following timing constraints:

- **Maximum Span**(ms): The maximum allowed time difference between the latest and earliest occurrences of events in the *entire* sequence.
- **Event-set Window Size**(ws): The maximum allowed time difference between the

latest and earliest occurrences of events in any *event-set*.

- **Maximum Gap**(xg): The maximum allowed time difference between the latest occurrence of an event in an event-set and the earliest occurrence of an event in its immediately preceding event-set.

- **Minimum Gap**(ng): The minimum required time difference between the earliest occurrence of an event in an event-set and the latest occurrence of an event in its immediately preceding event-set.

We assume the interestingness of a sequence to be defined based on how many times it occurs in the input data; i.e. its support. If the support is greater than a user-specified *support threshold*, then the sequence is called *frequent* or *interesting*. The the number of occurrences of a sequence can be computed in many ways, which are illustrated using the example shown in in Figure 3(a). The method COBJ counts at most one occurrence of a sequence for every object, as long as it is found within the given timing constraints. In the example, (1)(2) has two occurrences, one for each object. This method may not capture the sequences which are exhibited many times within a single object, which could really determine its interestingness. In the method CWIN, the support of a sequence is equal to the number of span-size windows it appears in. Each span-size window has a duration of ms, and consecutive windows have an overlap of $ms - 1$ units. Windows can span across a single object; i.e., no window can span across multiple objects. The support is added over all objects to get final support for a sequence. As shown in Figure 3(b), sequence (1)(2) has support of 3 for Object A, because it occurs in windows starting at time-points 0, 1, and 2. For object B, it occurs in 5 windows, hence the total support is 8. In other counting methods, instead of counting the span-windows, actual occurrences of a sequence are counted. Two options CDIST and CDIST_O are illustrated in Figure 3(c) and Figure 3(d), respectively. In CDIST, an event-timestamp pair is considered at most once in counting occurrences of a given sequence. So, there is only 1 occurrence of (1)(2) for Object A in the example, because there is no corresponding event 2's occurrence for event 1@2, 2@4 was used up in first occurrence. In CDIST_O, the occurrences are counted such that each new occurrence found has at least one different event-timestamp pair than previously found occurrences. So, (1)(2) has 3 occurrences for object B, and total of 5 occurrences, using this method.

The choice of which counting method to use is dependent on the problem and the judgment of the person using the discovery tool. For the purpose of our discussion in this paper, we will assume the method depicted in part (b), which counts the number of span-windows, because it is fairly general in the way it assigns interestingness to a sequence (especially when compared to method in part (a)).

The definition of sequential association presented above is a special case of the generalized universal sequential patterns described in [JKK99]. It combines the notions of generalized sequential patterns (GSPs) proposed in [SA96] and episodes proposed in [MTV97]. The GSPs do not have the ms constraint and counts according to method COBJ. The episodes are counted in just the same way as CWIN, but they do not have the xg, ng, and ws constraints. Also, the formulation presented in [MTV97] assumes presence of only one object. We believe

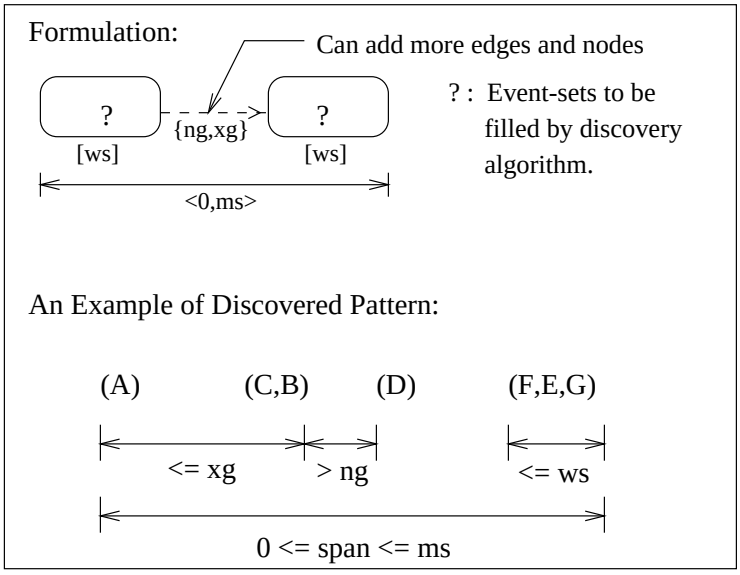


Figure 2: Generalized Formulation of Sequential Patterns

that our notion of sequential associations is fairly general for a wide variety of sequential data.

2.2 Serial algorithms for Sequential Associations

The complexity of discovering frequent sequences is much more than the complexity of mining non-sequential associations. To get an idea, the maximum number of sequences having k events is $O(m^k 2^{k-1})$, where m is the total number of events in the input data. Compare this to the $\binom{m}{k}$ possible item-sets of size k. Using the definition of interestingness of a sequence, and the timing constraints imposed on the events occurring in a sequence, many of these sequences can be pruned. If a sequence does not occur sufficient number of times or if it does not satisfy the timing constraints, it will not be present in the final result. But in order to contain the computational complexity, the search space needs to be traversed in a manner that searches only those sequences that would potentially satisfy both the support and timing constraints. As was done in the Apriori algorithm for non-sequential associations, the sequence discovery process can be guided by using the monotonicity property of support of a sequence. This property implies that as we add more events to the sequence, its support will decrease monotonously. So, if the frequent sequences having only one event are found first, then these can be used to build potential sequences having two events. All events occurring in such potential sequences are themselves frequent. Taking this notion further in an inductive manner, the frequent sequences having $k-1$ events can be used to build *potential* frequent sequences having k events. A potential k-sequence will be such that all its $(k-1)$-subsequences are themselves frequent.

The procedure outlined above is used in the GSP algorithm [SA96], but it discovers restrictive sequential patterns defined in [SA96]. We present below, a modified GSP algo-

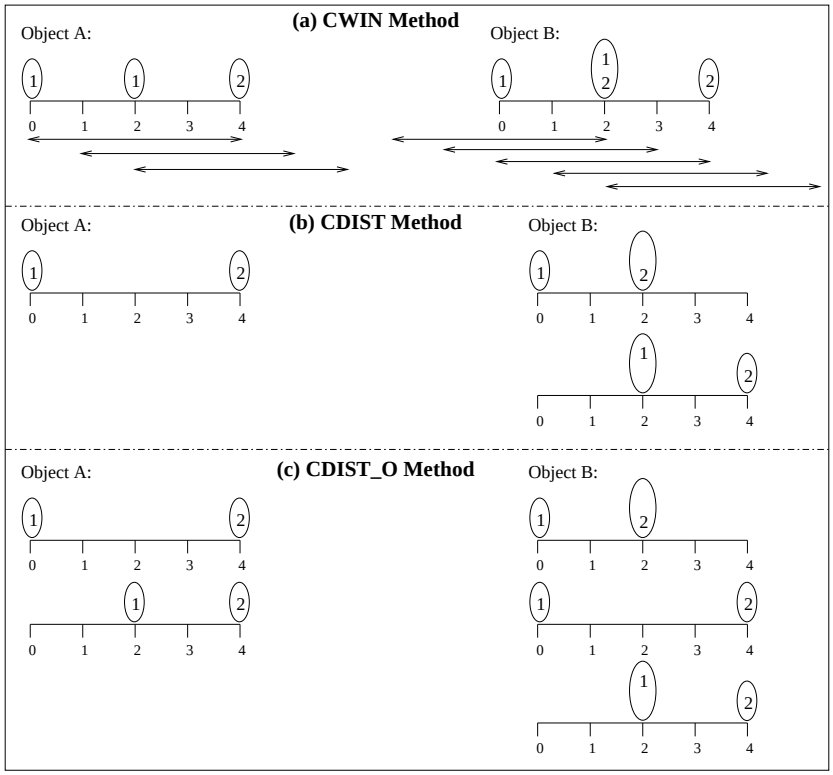


Figure 3: Illustration of Methods of Counting Support

rithm that discovers patterns according to our generalized sequential associations defined in previous subsection.

Similar to GSP, the algorithm works in iterations over the number of events in the sequence. In every iteration it has two phases. The first phase of join-and-prune generates the potentially frequent k-sequences, called candidates, from frequent $(k-1)$-sequences. The second phase counts the occurrences of candidates in object time-lines, and forms the set of frequent k-sequences, which seeds the next iteration. The structure of this algorithm is given in Figure 4.

As an illustration of the join-and-prune phase of the algorithm, consider the following example. Let the set of frequent 2-sequences be $F_2 = \{(1)(2), (1)(3), (1\ 3), (2\ 3), (3)(4)\}$. Now, in the join phase, (1)(2) will join with (2 3) to give a potential 3-sequence (1)(2 3). This sequence will not be pruned away because all of its 2-subsequences (1)(2), (1)(3), and (2 3) are in F_2. Now, consider the join between (1)(3) and (3)(4). This will generate a sequence (1)(3)(4), but it will be pruned away because (1)(4) is not present in F_2. This example assumes that the xg constraint is absent. If xg was specified, it would not be correct to prune (1)(3)(4) based on the absence of (1)(4), because it is not necessary for (1)(4) to satisfy the xg timing constraint in order for (1)(3)(4) to satisfy it.

The candidates that are generated after the join-and-prune phase need to searched in the input sequence data to count their occurrences. One simple way is to scan each object once for the occurrence of each candidate. This may become very expensive if the number of candidates is large, and if the number of of events occurring on input objects is large. Another possibility is to generate all the k-sequences present in all the span-size windows of an object's timeline and see if they are present in the set of candidates generated. This approach can also be very expensive especially because it ignores the information obtained from the previous pass that counted $(k-1)$-sequences in the timeline. Such information is implicitly stored in the set of candidate k-sequences generated by the join-and-prune phase. An efficient way to search occurrences of candidates is by storing them in a hash tree structure similar to the Apriori algorithm for non-sequential associations. An example of a hash tree is shown in Figure 5. A candidate is inserted into the hash tree by hashing on the events in that candidate. i^{th} event in the candidate is hashed at i^{th} level of the tree, where root is considered to be at first level. For example, while storing (1)(5)(9), event 1 is hashed at the root node N1, which leads to the leftmost child N2, where event 5 is hashed to reach its middle child N3, and finally event 9 is hashed to reach the rightmost leaf child of node N3. The depth of the hash tree and the number of internal nodes in the hash tree can be controlled by tuning the size of the hash table at internal nodes and the maximum number of candidates allowed to be stored at a leaf node.

The advantage gained out of constructing a hash tree is that the timeline of an object can be streamed through the branches of hash tree to identify only those candidates that could potentially occur in that object's timeline. An example of streaming a timeline is shown in Figure 5. The timeline is streamed through the hash tree by assuming that any event in the timeline can be the potential first event of a sequence, hence each event is hashed at the root node. The next event to hash is found using the timing constraints. As an example, let $ms = 10$, $xg = 9$, and $ws = 0$ be the timing constraints. If event 1 occurring at time

Structure of the Algorithm: (Based on GSP [SA96])

```
form Set of Frequent Sequences, F_1 each having 1 event;
k = 2;
while ( F_{k-1} is not empty )
      C_k = join-and-prune(F_{k-1}). C_k is the set of Candidates having k events
            [Store C_k in a hash tree access structure]

      for each object's timeline
           repeat
                [hash-tree-traversal]
                      stream the timeline through hash tree to search presence
                      of possible candidates, until leaf node is encountered; ⟵
                [leaf-count]
                     count the occurrences of candidates at leaf nodes;      ⟵
           until (no more traversing is possible)
      end

      form F_k from C_k by retaining only the large candidates (having count above
      support threshold);
end

function join-and-prune(F_{k-1}): returning C_k
begin
      if (k = 2) then
           for every pair of events, e_1 and e_2 in F_1,
                form c = (e_1)(e_2); add c to C_k
           for every pair of events, e_1 and e_2 in F_1, such that e_1 < e_2
                form c = (e_1 e_2); add c to C_k
      else
          for every c_1 in F_{k-1}
               search c_2 in F_{k-1} such that
                      deleting first event of c_1 and last event of c_2 yield same subsequences
               for every such c_2, form k-sequence c such that
                     c = (c_1)(e) if (e) is the last event-set of c_2; otherwise c = (c_1 e)
                           [e is the last event of c_2]
                     if every (k − 1) subsequence of c is present in F_{k-1}, then
                          add c to C_k
          end
      end
end
```

Figure 4: Structure of the Algorithm. Similar to GSP. The phases where modification is made are shown with an arrow (⟵).

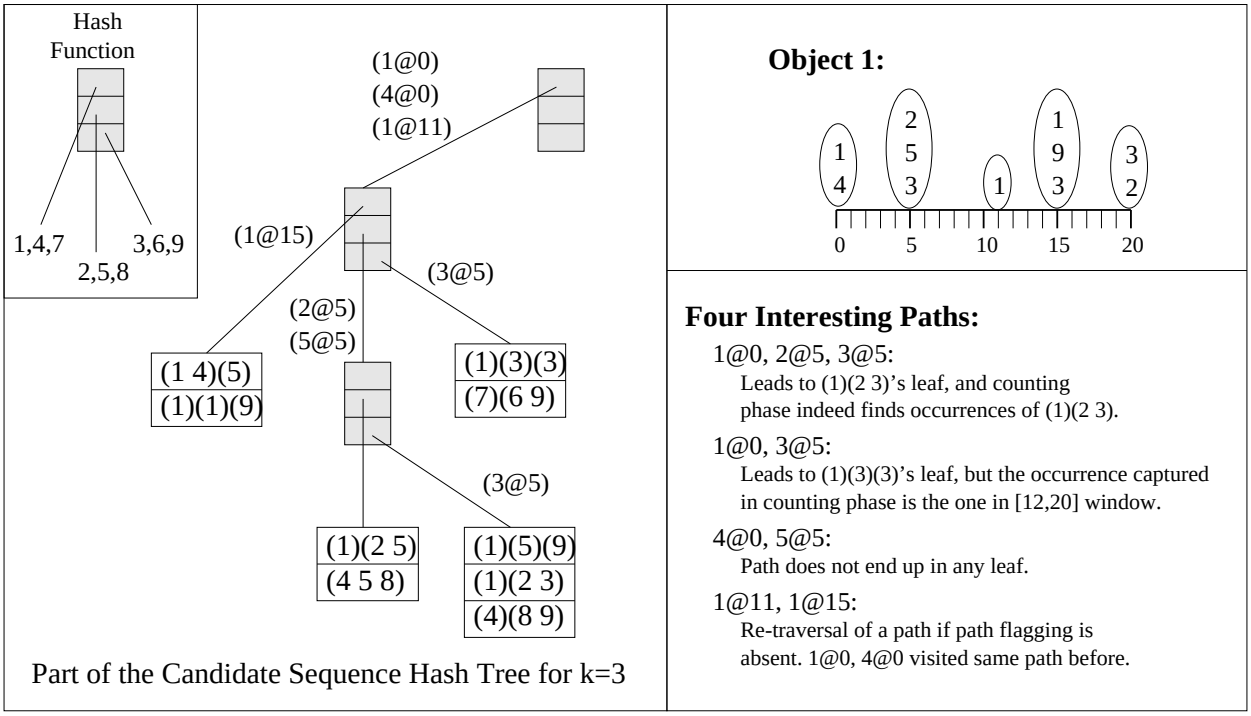


Figure 5: Illustrating the candidate hash tree, and the concept of streaming an object's timeline through the hash tree. The significance of using a hash tree can be understood by studying the four interesting paths listed.

0 (abbreviated as 1@0) is hashed at the root, then the only events eligible to be hashed at node N2 are 4@0, 2@5, 3@5, and 5@5. After hashing 5@5 at N2, although the event 1@12 satisfies the xg constraint, it does not satisfy the ms constraint, so it is not hashed at N3. Formally, if an event hashed at the root node occurs at time t_0, and if an event occurring at time t is just hashed at some node, then the event that is eligible to be hashed next must occur in the time-window defined by $[t - ws, \min(t_0 + ms, \max(t + xg, t + ws))]$.

In the process of streaming, when a timeline for an object reaches a leaf node, the algorithm counts all the occurrences of all the candidates stored at that leaf. Remember that we have defined the number of occurrences to be the number of span-size windows in which a sequence occurs. The first occurrence of a sequence is found using the method described in [SA96]. This search is based on searching event-sets in the order they appear in the sequence. The ws constraint is checked while searching an individual event-set. If an event-set is not found, search terminates. If event-set is found but it does not satisfy the ng and xg constraints with respect to the previous event-set, then a search is performed in the reverse direction to find occurrences of all previous event-sets within the ng and xg constraints. If the event-set is found, but is over the ms limit, then a new search is started from time $t_e - ms$, where t_e is the time of the latest event occurrence in the recently found event-set.

After finding the first occurrence, instead of stopping the search as done in GSP [SA96], our algorithm continues to search for the next occurrence. This next search starts at time $t + 1$, where t is the time at which the earliest event occurred in previously found sequence. Let the first occurrence of a sequence be found in the range $[t_s, t_e]$. Then the number of span-windows, that this occurrence contributes to, is equal to $(t_s - (t_e - ms + 1))$. Now, the next sequence search starts at time $t_s + 1$. If the next occurrence is found in time range $[t_s^{'}, t_e^{'}]$, then the number of span-windows that this occurrence contributes to is equal to $(t_s^{'} - max(t_e^{'} - ms, t_s + 1)]$; i.e. count all the span-windows that start after the time t_s and lie within ms duration of $t_e^{'}$. As an example, consider the object 1's timeline shown in Figure 5. The constraint ms is 10. The first occurrence of (1)(2 3) is found in the range [0,5], contributing to all the span-windows that which start at time-points between -4 and 0. The next occurrence of (1) (2 3) is found in the range [11,20]. This contributes to the windows starting at time-points 11 and 12. The next occurrence is found to be in the range [15,20], this accounts for the windows starting at time-points 13, 14, and 15. This discussion is relevant when CWIN method is used for counting. For other methods, appropriate changes need to be made. For example, when CDIST method is used, each event-timestamp pair must be flagged to indicate if it has been used towards counting the occurrence of a given sequence. When CDIST_O method is used, the counting process is similar to CWIN, but instead of counting the windows, just one occurrence is added each time a search succeeds.

Following points should be noted regarding the algorithm discussed above.

- The role of a hash tree to increase the efficiency of search can be fulfilled only if the hash tree is not too deep and/or too large, because in the worst case the amount of work involved in streaming a timeline through the hash tree is equal to the amount of work done in finding first occurrence of each candidate using a straight-forward search (when every candidate is stored at a different leaf in the hash tree). On the other

hand, if the number of candidates is very large *and* the tree is very shallow or small, then the advantage of building a hash tree would be lost again. Hence, the size of the hash table at each node and the maximum number of candidates allowed to be stored at leaf nodes play an important role in determining the efficiency of the algorithm.

- If an object's timeline is very large, then the operation of streaming a timeline needs to be optimized to flag the paths which are impossible and the root-to-leaf paths which are already traversed. This avoids their repeated traversal.

- If an object's timeline is very large, then the counting operation at the leaf can possibly take a very long time because the entire timeline would need to be traversed for each candidate at every leaf reached.

3 Parallel Formulation: Issues, Challenges, and Some Solutions

If the input sequence data has following features, then the points mentioned at the end of previous section depict the limitations of serial algorithms[1].

- Enormity; i.e., large number of objects and/or large time-lines for many objects. Serial algorithms would take a very long time to in the counting phase for such datasets.

- High dimensionality; i.e., large number of events. The number of candidates generated for such datasets will be very large; hence, either they may not fit in the memory available for a single processor, or they would make the hash tree data structures act counter-productively if their size and structure is not optimally managed.

This motivates the need for parallel formulations. In this section, we will briefly discuss the issues and research challenges involved in developing effective parallel formulations of sequential pattern discovery algorithm.

The parallel formulation should be able to divide two entities among processors. One is the computational work and other is the memory requirement. These should be divided such that the time and memory limitations faced by serial algorithms could be minimized, and it should be possible to achieve this with as little overhead as possible. In parallel formulations, the overheads come mainly from load imbalance (causing idling of processors) and the communication required to co-ordinate the computations performed by different processors.

The computational load in sequential pattern discovery algorithm consists of candidate generation and counting of candidates. The memory requirements come from storing the

[1]The terms serial and sequential should not be confused. Traditionally, sequential and serial are both used to describe algorithms that would run on single processor machines. Here, we use the term serial to represent such algorithms, and reserve the term sequential to indicate the temporal or sequential nature of the input data

input datasets and the candidates generated. Depending on how the candidates and object time-lines are distributed among processors, different parallel algorithms are possible. One set of parallel algorithms is given in [SK98]. These algorithms assume the sequential pattern format given in [SA96]; hence, their algorithms do not have concepts of span (ms) and counting multiple occurrences of a sequence in a single timeline. Also they assume a market transaction type of dataset, in which the object time-lines are usually very short. So, their formulation distributes the transaction database as discussed above in the first possibility of distributing objects. Then they use straight-forward extensions of their NPA, SPA, HPA algorithms for non-sequential associations to get algorithms which do not partition the candidates (NPSPM), or partition the candidates in a simple round-robin manner (SPSPM) or in a more intelligent manner using hash functions (HPSPM). The last formulation, HPSPM, is similar in concept to the idea discussed above for partitioning candidates for use with the hash join method. The way the count phase is performed in NPSPM is similar to the CD algorithm for non-sequential associations [HKK97]. The occurrences for all candidates are first counted in the local dataset, and then they are broadcast to all other processors to determine the set of frequent sequences. The counting in SPSPM is performed in a way similar to the DD algorithm for non-sequential associations, where every object's timeline is sent to every processor. HPSPM, in k^{th} iteration, generates all k-sequences present in each object's timeline and hashes them using the same hash function as was used for hashing the candidates to distribute them among processors. Each sequence is sent to the processor it hashes to, and searched for in the list of candidates stored there. The HPSPM algorithm is shown to perform better than the rest two, but it also faces severe limitations when the object time-lines are very large, and when it is extended to use the counting method used in our generalized sequential pattern formulation.

In the following, we present several parallel formulations of our own, that take into account the generalized nature of sequential patterns. The intention is to bring out the challenges involved in designing effective parallel formulations.

In the first formulation called EVE (event distribution), we distribute input data and replicate the candidate hash tree on all the processors. The candidate generation phase is done serially on all the processors. Three different variations of EVE algorithm are presented to cater to different scenarios emerging depending on the number of objects, the length of the time-lines in terms of the number of events happening on them, and the value of ms. After presenting these variations, we will present an algorithm EVECAN, which distributes events as well as candidates among processors, to overcome some of the problems that EVE might face.

3.1 EVE-S: Simple Event Distribution Algorithm

For shorter time-lines and relatively large number of objects, the input data is distributed such that the total number of event points is as evenly distributed as possible within the constraint that a processor gets all the entire timeline of every object allocated to it. In this case, the algorithm is similar to the NPSPM algorithm discussed before. It is embarrassingly parallel as far as counting phase is concerned, except for the final communication operation

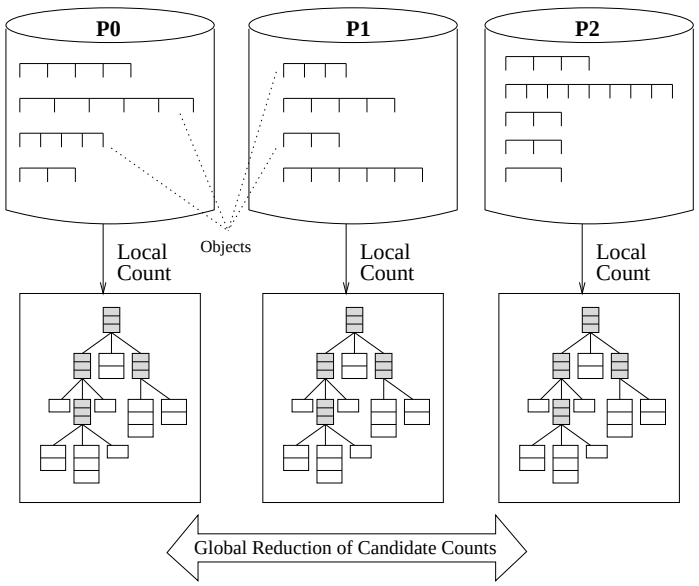


Figure 6: Illustration of EVE-S algorithm.

required to accumulate the candidate counts. EVE-S is illustrated in Figure 6.

3.2 EVE-R: Event distribution with partial data replication

This formulation is designed for the scenario in which there are relatively small number of objects (less than the number of processors), each object has a large timeline, and the span value (ms) is relatively small. The input data is distributed as follows. The timeline for each object is split across different processors such that the total number of events assigned to different processors is similar. Note that the sequence occurrences are computed in span-size windows. We assume that the span value is small such that no span window spans across more than two processors. But, still one processor will have some span-windows that do not have sufficient data to declare the occurrence of an sequence in them. This is resolved in EVE-R by gathering such missing data from neighboring processors. Each processor gathers data that is required to process the last span-window beginning on that processor. This is illustrated in Figure 7. Since we assume that span-windows do not straddle more than two processors, just the neighbor-to-neighbor communication is sufficient. Once every span-window is complete on all processors, each processor processes only those span-windows which begin at the events points originally assigned to it. For example, processor P0 processes windows that begin at time instances 0, 1, 2, and 3, whereas processor P1 will process windows that begin at 4, 5, 6, and 7. By distributing the event points equitably, load balance can be achieved. As in EVE-S algorithm, the occurrences are collected by a global communication (reduction) operation, in the end.

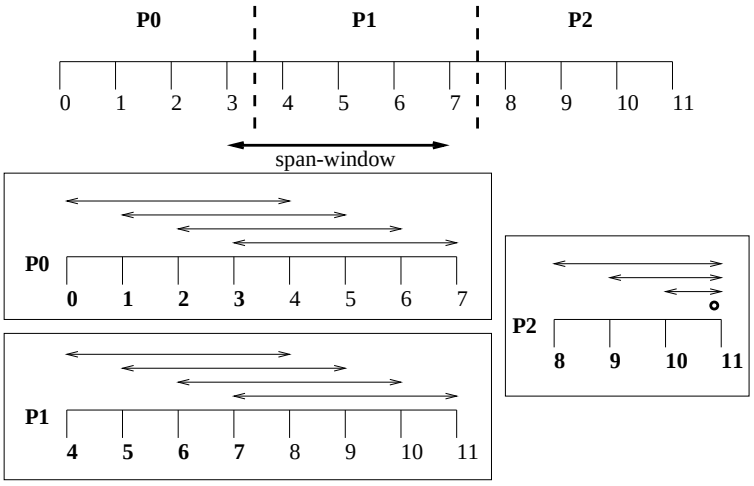


Figure 7: Illustration of EVE-R algorithm.

3.3 EVE-C: Complex Event Distribution Algorithm

This formulation depicts the most complex scenario as far as distribution of the counting workload is concerned. This happens when there are small number of objects, each object has a large timeline, and the span value is large such that after splitting the object time-lines across processors, the span-windows straddle more than two processors. There are two ways to handle this.

One way is to replicate the data across processors such that no processor has any incomplete or partial span-window. This is the same idea used in EVE-R, what makes it different is the fact that the amount of replication can become very large in this case. So, if processors do not have enough disk space to hold the entire replicated information, this approach may not be feasible. Even when there is enough disk space available on each processor, the replication of data may result in a lot of replication of work. This is illustrated using a simple example in Figure 8. Consider counting the number of occurrences for the sequence (3)(5). If the entire span-window [1,5] was available on a single processor, the serial algorithm would just count a single occurrence of (3)(5) in the range [4,5] and increment its count by 4 in one single operation. Now, let the span-window be split across processors, and it is replicated on processors P0, P1, and P2 as done in EVE-R. Each processor is assigned to count occurrences of (3)(5) in windows beginning at their respective event points. Processor P0 will search for an occurrence of (3)(5) in window beginning at 1, similarly P1 will search for occurrence of (3)(5) for windows beginning at 2 and 3. Thus, the occurrence of (3)(5) is being searched by all the three processors resulting in replication of work. In order to avoid this replication of work, a processor can be restricted to find only those occurrences which have at least one of the events assigned to that processor. With this restriction, in our example, P0 and P1 do not count (3)(5)'s occurrence, only P2 does. For (3)(5), P2 can find the correct count by using the same formula as a serial algorithm would use. But, consider the case of counting (1)(3)(5). With the restriction just imposed, if occurrences are counted using the formula of a serial algorithm, P0 counts for occurrence in window starting at 1, and P2 counts for

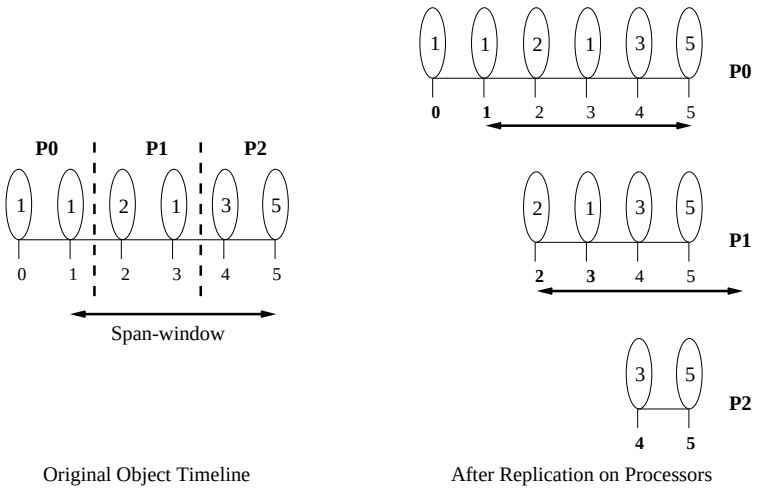


Figure 8: Illustration of EVE-C algorithm when data is replicated across processors.

occurrences in windows starting at 1 and 2! Thus, there is a danger of double counting. If P1 communicates its range of occurrence of (1)(3)(5) to P0 (or vice versa), incorrect counting can be avoided. In general, a processor would have to communicate information for all the occurrences that might belong to windows starting on other processors. This may result in a lot of communication, especially when the number of candidates is large.

Thus, when data is replicated, there is trade-off between the approach of replicating the work with no communication cost (except for the data replication cost), and the approach of avoiding work replication by paying the extra cost of communicating the occurrences.

The second way to handle this is not to replicate the data. Now, two kinds of situations need to be handled. We will use the timeline shown in Figure 9(a) to illustrate this.

- Those occurrences that are found completely on a single processor might contribute to span-windows that begin on other processors. For example, (1 4) is found on both P0 and P1 in Figure 9(a). Care should be taken to avoid counting (1 4) twice in the [1,9] window. This can be done by simply restricting each processor to count for only those windows that begin at event-points assigned to it. For example, P1 will associate (1 4)'s occurrence only with windows beginning at 4 and 5. But, if this is done, the counting algorithm could miss counting occurrences in some span-windows that begin on other processors. For example, (1 4) also occurs in windows beginning at 2 and 3, but this will not be accounted for using above strategy. In this case, P1 should communicate the range of its (1 4) occurrence to P0 so as to account for windows beginning at 2 and 3. As can be seen, for such occurrences, the situation is similar to the one encountered while avoiding replication of work with data replication approach.

- Some occurrences cannot be declared to occur in some span-windows because there may not be sufficient data available on a single processor. For example, occurrence of (2)(1)(2) cannot be found unless all processors co-operate. This gives rise to the most complex method of parallelizing the counting process.

The first phase of hash tree traversal determines which leaves of the tree are reachable by a given timeline. The traversal would reach an internal node, and realize that it cannot find next eligible event to hash because the event may lie on the next processor. In such case, instead of terminating the search on that path, the search is suspended. The state of suspension is stored at every such node. All the suspended states need to be passed to the processor having the next consecutive segment of the timeline. The receiving processor will start traversing the hash tree from the suspended states that it receives. Using this procedure, each processor will find a set of leaves that are reachable by the timeline. This set needs to be broadcast to all the processors.

Once every processor knows the set of candidates that needs to be actually searched for in the timeline, the second part of the counting phase starts. Remember the way counting happens serially. Occurrences of sequences are searched one after another, starting from the smallest time-point in the timeline. The number of span-windows that each occurrence contributes to is counted based on the range in which that occurrence is found and the starting time of the previously found occurrence. In order to do this in parallel for partial occurrences, a processor needs to transfer all its partially found occurrences along with their timestamps to the next processor (which has the next consecutive part of the timeline). The serial (chain-like) nature of this procedure is inevitable. The worst case is when each processor finds all its partial occurrences before communicating them to the next processor, in which case the parallelism achieved for the these occurrences is minimal. This can be slightly alleviated by sending some partial occurrences before finding next, in a pipeline manner, with the hope of achieving some overlap of communication and computation.

This entire approach can become very expensive if any of the following happens:

1. The amount of partial work transferred, both in the hash tree traversal phase (suspended hash tree states) and in the counting phase (partial occurrences), can quickly become very large. This is due to the nature of the association discovery problem in which each span window has a potential of supporting exponential number of sequences. The cost arising due to performing hash tree traversal phase in parallel can be avoided by just deciding not to build any hash tree access structure. But, that might increase amount of partial sequence occurrences that need to be sent in counting phase. So, the decision of whether to build the hash tree would depend on the number of candidates found. For some earlier passes, building a hash tree might pay off by reducing communication in the counting phase.
2. In the counting phase, if a search for a sequence needs to be backtracked, then the formulation becomes much more complex. There needs to be a back and forth communication between processors to do backtracking in parallel. This can be avoided by making a processor search for all possible occurrences of each event-set in the sequence and passing them onto the next processor. But, this would imply an enormous amount of work to be transferred, much more than what was eluded to in the first point above.

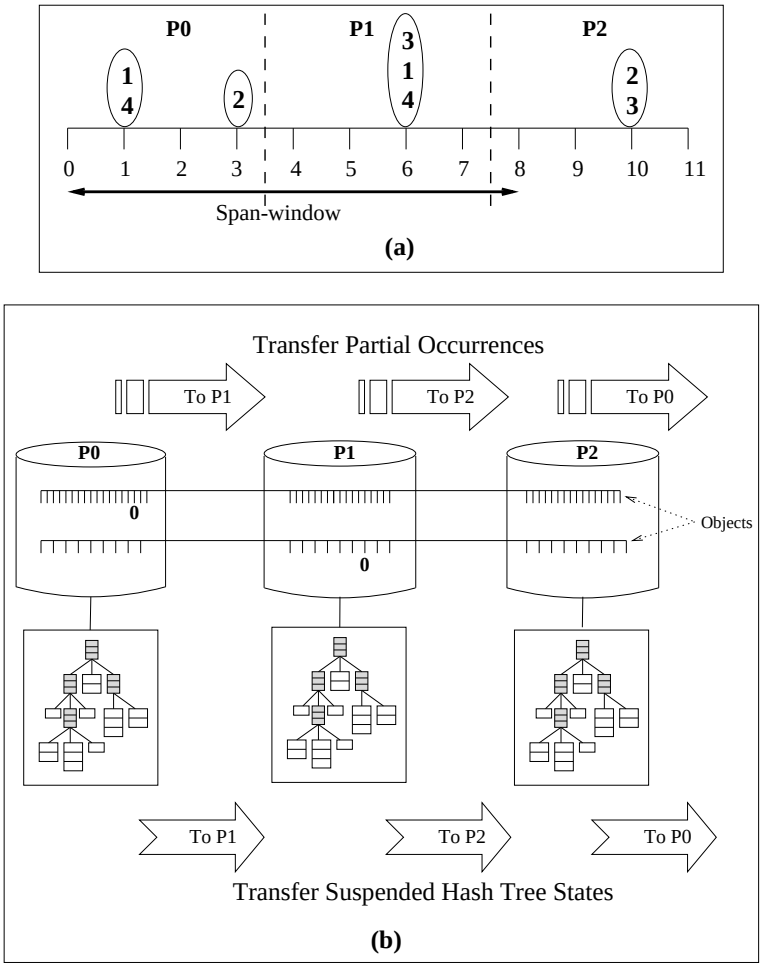


Figure 9: Illustration of EVE-C algorithm when data is not replicated across processors. (a) An example timeline used to illustrate issues in counting occurrences, (b). A schematic of the process used to handle partial occurrences.

This leads us to consider a trade-off between the choices of replicating and not replicating the input dataset. The scenarios described above, in which the formulation with no replication of data can become very expensive, are not rare. So, we believe that in many cases, the formulation which replicates the data on all processors might be desirable for lower complexity as well as better execution time. The worst case cost of having the entire dataset residing on all processors, and incurring either the work replication cost or the cost of communicating occurrence ranges, might be much less than the cost of transferring partial work. Moreover, we believe that the computational concurrency extracted by the replication approach could be more than the non-replication approach.

Finally, a point to note is that the above description assumed the CWIN counting method. With CDIST counting method, which requires each event-timestamp pair to be considered at most once for each candidate sequence, the parallelization becomes much more complex.

3.4 Event and Candidate Distribution (EVECAN) Algorithm

In the set of EVE algorithms described above, it is assumed that the candidates are replicated over all the processors. This may not be desirable when the number of candidates is very large, and with the complexity of sequential patterns such scenarios are not uncommon. Large number of candidates results in two things. The set of candidates may not fit in the memory of a processor, in which case the hash tree needs to be built in parts. This involves multiple I/O passes over the disk for counting the candidates. Secondly, since EVE algorithm builds candidates serially on all processors, thus losing out on extracting the possible concurrency, the amount of time spent in generating the large number of candidates can be significantly large.

These issues are addressed in our second formulation, called EVECAN (event and candidate distribution). In this algorithm, we partition the input data similar to EVE. But, now the candidates are also distributed. They are stored in a distributed hash table. The hashing criterion is designed to maintain equal number of candidates on all processors. One simple hash function can be based on the lexicographical ordering of candidates and splitting them among processors such that all candidates assigned to one processor have a common prefix sequence. The non-local candidates required for the join-and-prune phase are obtained using the scalable communication structure of the parallel hashing paradigm introduced in [JKK98]. Now since all the processors must count all the candidates, there are two options. In the first option, we keep the candidates stationary at processors and a local hash tree access structure is built for these candidates. The input data is circulated among processors in a fashion similar to that of the round-robin scheme proposed for IDD algorithm of [HKK99]. But this option may work only when the span value is small, in which case we will circulate the span-windows. For large span-values, it could be very expensive to send all the span-windows to all the processors. In such cases, second option can be used, which is to move around the candidates in a round robin fashion. Building a new hash tree for every set of candidates received can be prohibitively expensive, so no access structure may be built for this option. In both the options, a hash function is used to do a relatively quick search of whether a span-window can contain the candidates stored at that processor. Figure 10 pictorially depicts the EVECAN algorithm.

Note that, with the option of moving candidates around, the object time-lines can be distributed according the most complex scenario of EVE algorithm. When this happens, the possibility of using a EVE algorithm by itself (replicating the candidates over all processors) needs to be weighed against the possibility of using EVECAN algorithm. EVECAN would need to handle the large time-lines using method similar to EVE-C in every rotation of candidates. This can be very expensive.

Here, we have not presented any implementation details or experimental results for the EVE and EVECAN parallel formulations. This section was intended merely to bring out the importance of sequential pattern discovery, the need for designing parallel algorithms for discovery task, and the issues and challenges involved in designing efficient parallel formulations.

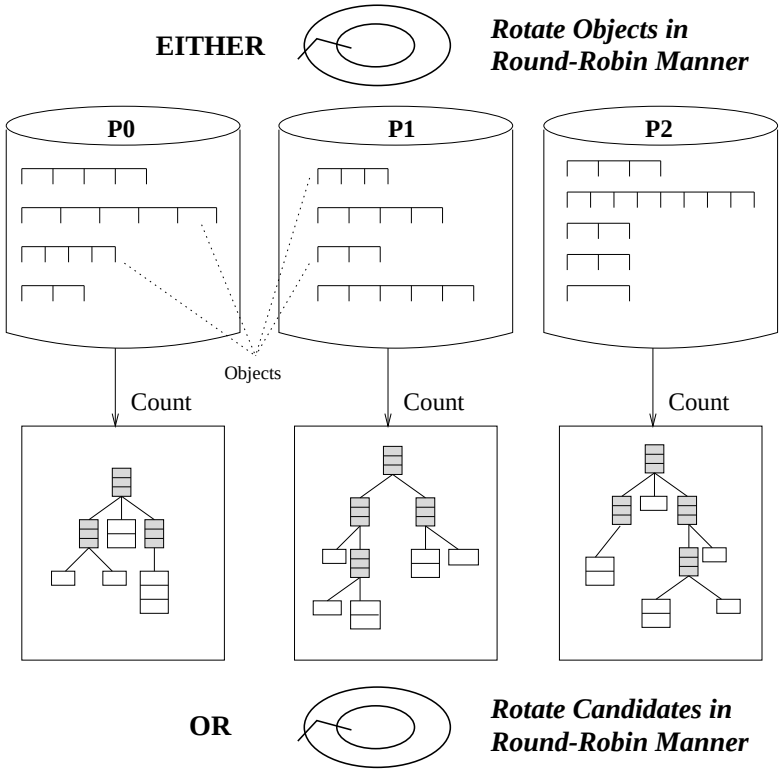


Figure 10: Illustration of EVECAN algorithm for parallel discovery of generalized sequential associations.

4 Conclusion

In this paper, we elaborated on the issues and challenges of parallelization, presented a comparative review of existing parallel algorithms, and proposed new parallel algorithms for mining generalized sequential associations.

We considered formulations that utilize the temporal and sequential information in the datasets to discover sequential associations. A generalized universal formulation was presented, which characterizes a sequential association by various timing constraints and counting strategies. Specifically, this formulation introduces the concept of maximum span, and allows counting for more than one occurrences in an object's time-line. The modification of existing GSP algorithm was presented for discovering these more general sequential associations. A case was made for the need of parallel formulations. The maximum span constraint and counting multiple occurrences in an object, give rise to challenging scenarios for designing effective parallel algorithms to discover generalized sequential associations. We elaborated on four different parallel formulations which work for datasets of different characteristics.

For datasets having large number of objects with smaller time-lines, a simple event distribution algorithm (*EVE-S*) is proposed. This algorithm is very similar to the *CD* approach of parallelizing non-sequential associations, except for the differences in the details of counting occurrences. The parallelization becomes challenging especially when the number of objects is smaller than the number of processors and each object has large and dense time-line. This situation can occur in many application domains depending on the formulations of objects and events in the input dataset (for example, in telecommunications data, if an object is the month in which the alarms are recorded, and events are pairs of switch and alarms happening on them during the given month). We discussed various issues in parallelizing under such circumstances. The formulation *EVE-R* tries to reduce the problem to EVE-S algorithm by replicating the data over such that each processor has complete information for the span-windows assigned to it. The data replication might become expensive when the span values are large (possibly extending to large portions of the time-line). This leads to the complex *EVE-C* formulation, where we discussed the pros and cons of replicating versus not replicating the data. It was shown that, in most situations, the large amount of work that needs to be shifted using the no-replication approach might quickly make it inefficient as compared to the replication-of-data approach even after paying the replication cost and cost of communicating the occurrence information among processors. Finally, a formulation *EVECAN* is presented using similar principles as the IDD parallel algorithm for non-sequential associations. It specifically handles the combinations of minimum support and datasets which generate large number of candidates. EVECAN is different from IDD in the way it introduces the movement of candidates, and in its parallelization of candidate generation phase by building and probing a distributed hash table of candidates using scalable communication mechanisms.

Overall, this paper serves as a comprehensive account of various design issues, challenges, and different parallelization strategies for mining sequential associations.

References

[AIS93] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. of 1993 ACM-SIGMOD Int. Conf. on Management of Data*, Washington, D.C., 1993.

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th VLDB Conference*, pages 487–499, Santiago, Chile, 1994.

[AS96] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. of the Int'l Conference on Data Engineering (ICDE)*, Taipei, Taiwan, March 1996.

[CHY96] M.S. Chen, J. Han, and P.S. Yu. Data mining: An overview from database perspective. *IEEE Transactions on Knowledge and Data Eng.*, 8(6):866–883, December 1996.

[HKK97] E.H. 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*, Tucson, Arizona, 1997.

[HKK99] E.H. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. *IEEE Transactions on Knowledge and Data Eng. (accepted for publication)*, 1999.

[JKK98] Mahesh V. Joshi, G. Karypis, and V. Kumar. ScalParC: A new scalable and efficient parallel classification algorithm for mining large datasets. In *Proc. of the 12th International Parallel Processing Symposium*, Orlando, Florida, April 1998.

[JKK99] Mahesh V. Joshi, George Karypis, and Vipin Kumar. Universal formulation of sequential patterns. Technical Report Under Preparation, Department of Computer Science, University of Minnesota, Minneapolis, 1999.

[MTV95] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering frequent episodes in sequences. In *Proc. of the First Int'l Conference on Knowledge Discovery and Data Mining*, pages 210–215, Montreal, Quebec, 1995.

[MTV97] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. Technical Report C-1997-15, Department of Computer Science, University of Helsinki, Finland, 1997.

[SA96] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. of the Fifth Int'l Conference on Extending Database Technology*, Avignon, France, 1996.

[SK98] Takahiko Shintani and Masaru Kitsuregawa. Mining algorithms for sequential patterns in parallel: Hash based approach. In *Research and Development in Knowledge Discovery and Data Mining: Second Pacific-Asia Conference (PAKDD'98)*, pages 283–294, Melbourne, Australia, April 1998.