2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems

# Topology-Aware Event Sequence Mining for Understanding HPC System Behavior and Detecting Anomalies

Zongze Li and Song Fu
Department of Computer Science and Engineering
University of North Texas
Denton, Texas, USA
Zongzeli2@my.unt.edu, song.fu@unt.edu

Sean Blanchard and Michael Lang
HPC-DES Group; Computer, Computational, and Statistical Sciences Division
Los Alamos National Laboratory
Los Alamos, New Mexico, USA
seanb@lanl.gov, mlang@lanl.gov

***Abstract*— System logs provide invaluable resources for understanding system behavior and detecting anomalies on high performance computing (HPC) systems. As HPC systems continue to grow in both scale and complexity, the sheer volume of system logs and the complex interaction among system components make the traditional manual problem diagnosis and even automated line-by-line log analysis infeasible or ineffective. Sequence mining technologies aim to identify important patterns among a set of objects, which can help us discover regularity among events, detect anomalies, and predict events in HPC environments. The existing sequence mining algorithms are compute-intensive and inefficient to process the overwhelming number of system events which have complex interaction and dependency. In this paper, we present a novel, topology-aware sequence mining method (named TSM) and explore it for event analysis and anomaly detection on production HPC systems. TSM is resource-efficient and capable of producing long and complex event patterns from log messages, which makes TSM suitable for online monitoring and diagnosing of large-scale systems. We evaluate the performance of TSM using system logs collected from a production supercomputer. Experimental results show that TSM is highly efficient in identifying event sequences on single and multiple nodes without any prior knowledge. We apply verification functions and requirements and prove the correctness of the event patterns produced by TSM.**

***Keywords—High performance computing systems; System monitoring and diagnosis; Anomaly detection; Sequence mining; Event patterns.***

## I. INTRODUCTION

High performance computing (HPC) systems continue growing in both scale and complexity. For example, the fastest supercomputer, Summit at Oak Ridge National Laboratory, has 9,216 IBM POWER9 processors and 27,648 NVIDIA Volta V100s GPUs. The Trinity supercomputer (#6 in the Top500 list) at Los Alamos National Laboratory has more than 19,000 heterogeneous (i.e., Intel Haswell and Intel Knights Landing) compute nodes. These large-scale, heterogeneous systems generate tens of millions of log messages every day. In addition to the sheer volume, both the format and the content of these log messages vary dramatically, depending on system architecture, hardware configuration, management software, and type of applications. Effective log analysis for understanding system behavior and identifying system and component anomalies and failures is highly challenging.

Existing log analysis approaches focus on discovering distribution and precedence relation among log messages, they are not effective for discovering subtle behavior patterns and their transitions, and thus may overlook some critical anomalies. log messages and system events are not isolated from each other. In our recent research [18], we developed a *System Log Event Block Detection framework* (named SLEBD) which explores the probability of messages occurring together in a flexible period of time and leverages the law of total probability to consolidate messages that occur together even with variations into Event Block from system logs. Analysis at the event level can provide a richer semantics of system behaviors and thus enable to detect more subtle anomalies that the traditional line-by-line analysis methods cannot find.

*Sequence mining* aims to discover important patterns among a set of objects. It can help us discover regularity among events, detect anomalies, and predict events in HPC environments. Existing sequence mining methods, such as GSP [1], AprioriAll [2][3] and SPADE [4], are mostly based on the Apriori algorithm [17]. Apriori-like algorithms perform multiple rounds of scanning of objects to detect all possible pattern candidates, which is compute intensive and runs for a long time. Moreover, this type of algorithms generates a large number of candidate sequences which need a large database to store and analyze. The time and space complexity of Apriori-like algorithms makes them inefficient for processing objects with a large population and complex relation, such as huge logs and various events in production HPC systems. To address these issues, Han et al. proposed the FreeSpan [5] and PrefixSpan [6] algorithms which are still based on the Apriori method but avoid the expensive candidate generation process, that is they are spatially more efficient, but the time complexity is still prohibitive.

In this paper, we aim to detect anomalous system and component behaviors from a large number of events identified in HPC systems by developing a novel, *topology-aware sequence mining* (named TSM) method. Our previous studies of HPC systems have revealed that certain events commonly happen on compute nodes. Some events may appear multiple times in a period of time. However, a new execution sequence does not start until an old sequence of the same type is completed. As a result, the event sequence can be represented by a Directed Acyclic Graph [7]. This inspires us to design a sequence mining algorithm that leverages the topology [8] information. TSM identifies all possible long sequences among system and component events after generalizing their positions

978-1-7281-2058-4/19/$31.00 ©2019 IEEE
DOI 10.1109/HPCC/SmartCity/DSS.2019.00315

from massive events in a cost-effective manner. TSM is highly efficient as it scans events only once to collect the temporal and spatial information of events, and it does not require a large number of comparison and merge operations to generate sequences. In addition to the low time complexity, TSM is space efficient as it needs small memory and storage space to store the scanning results and temporary data. In our experiments, we use the produced event patterns to further detect system anomalies by leveraging recurrent neural network [13] models

.

## II. Related Work

Existing sequence mining methods are mostly Apriori-like [17]. An Apriori-like method, such as GSP [1], uses multiple candidate generation-and-mining scans and tests to produce all possible sequences. This process is both time-consuming and space-inefficient. To address this issue, Han et al. proposed FreeSpan [5]. FreeSpan scans target objects to generate length-2 subsequences. It then projects length-2 subsequences to length-3 subsequences and continues until no more sequences of longer length can be projected from shorter subsequences, e.g.,

[A, B], [B, C], [A, C] => [A, B, C]

A length-$N$ sequence is projected from $N$ length-($N$-1) subsequences, which requires $N$ to $\sum_{i=1}^{N} i$ times of subsequence comparisons and merges. As an example, Table I shows a simple set of events.

TABLE I. Example of Events

| Sequence Id | Sequence |
|---|---|
| Seq_1 | [A, B, C, A, B, C, A, B, C] |
| Seq_2 | [A, B, C, A, B, C, A, B, C] |

FreeSpan can produce 325 length-1 to length-9 subsequences. Most of the sequences are subsequences of other sequences. However, there is only one useful length-3 sequence, i.e., [A, B, C] in this example. In our experiments on an HPC system, one event sequence contains more than 80 events. In the worst case, FreeSpan needs to generate 3,160 length-2 subsequences and 82,160 length-3 subsequences for a length-80 sequence.

To reduce the large number of subsequences generated by FreeSpan, Han et al. developed an improved algorithm, called PrefixSpan [6], which treats some subsequences as prefixes and only projects subsequences of longer length based on the prefix subsequences. The PrefixSpan algorithm has been used in many areas, for example, analysis of supermarket records [9].

A critical problem with Apriori clustering is that objects clustered into one class must have the same number of occurrences. In the HPC system, however, events do not happen for the same number of times. One event could be followed by a repeated event and vice versa. Thus, Apriori-like algorithms are not suitable for event analysis in HPC environments.

The preceding issues require us to develop a new sequence mining method which can discover execution sequences from various events on HPC systems. In this paper, we propose a topology-aware sequence mining (TSM) algorithm to address the following major problems in the existing sequence mining methods.

- They generate a huge number of temporary, short subsequences -- high space complexity.
- They perform a large number of comparison operations and merging short subsequences into longer sequences -- high time complexity.
- They produce many misleading subsequences which are not useful for behavior analysis and anomaly detection for HPC systems.

## III. Overview of System Log Event Block Detection

In [18], we presented a System Log Event Block Detection (SLEBD) framework. It can convert the lengthy and unstructured messages in system logs into a compact and structured list of event blocks. Thus, the complexity of log analysis can be significantly reduced and the results are more interpretable and easier to understand.

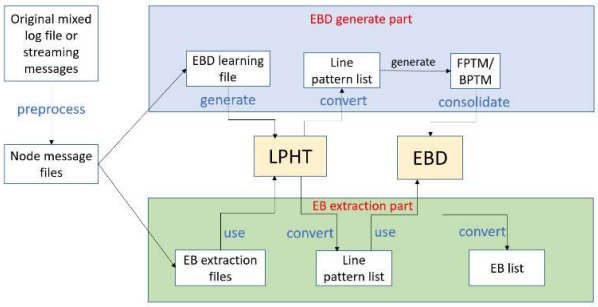


Figure 1. Major components and workflow of SLEBD

Figure 1 shows the architecture of SLEBD. Logs collected from HPC systems are first preprocessed by separating messages into multiple files based on node IDs. SLEBD then uses create a single line pattern for each log message and stores unique line patterns in a Line Pattern Hash Table (LPHT). Each line pattern has an identifier in the form of "[LinePattern_$num]". A line pattern is created by alphabetic words in the message and their corresponding positions. Numbers are treated as variables, and not included in line patterns. The following is an example showing the line pattern for a log message extracted from Mutrino's system logs.

2015-02-13T13:16:45.462890-06:00 c0-0c0s0n1 ACPI: PCI Root Bridge [UNC0] (domain 0000 [bus ff])

**A syslog message from Mutrino supercomputer**

[[0, ACPI:], [1, PCI], [2, Root], [3, Bridge], [5, (domain), [6, [bus]]

**A single line pattern**

Line pattern lists for different nodes are co-analyzed to determine how often every pattern occurs and what adjacent line patterns happen before (called backward patterns) and after (called forward patterns) the line pattern in question. A

Forward Probability Transition Matrix (FPTM) and a Backward Probability Transition Matrix (BPTM) are generated. The Law of Total Probability is applied to determine if two line patterns occur together with high confidence using FPTM and BPTM. The consolidated event block patterns are stored in an Event Block Database (EBD) which is used to identify and extract event blocks from system logs.

## IV. Mining Event Patterns Based on Topology

### A. Topology position between events in sequences

We generalize five possible topology positions between two events in a sequence:

1. *Prior*: Event_A always occurs prior to Event_B.
2. *After*: Event_A always occurs after Event_B.
3. *Wrapping*: Event_A always occurs both prior to and after Event_B, i.e., Event_A wraps around Event_B.
4. *Wrapped*: Event_A always occurs between two Event_Bs, i.e., Event_B is wrapped by Event_A.
5. *Tangling*: sometimes Event_A occurs prior to Event_B and sometimes Event_A occurs after Event_B.

Table II shows an example of event sequences.

TABLE II. Event Sequence Example

| Sequence_id | Sequence |
|---|---|
| Seq_1 | [A, B, C, D, E, C] |
| Seq_2 | [B, A, C, D, E, C] |
| Seq_3 | [B, A, D, E] |
| Seq_4 | [B, C, D, E, C] |

From these two sequences, we can see

1. Events A and B are *prior* to events C, D and E.
2. Events C, D, and E are *after* events A and B.
3. Event C *wraps* events D and E.
4. Events D and E are *wrapped* by event C.
5. Event A and event B are *tangling*.

### B. Generating prior transition matrix and position status matrix from sequences.

TSM scans sequences once and produces a Prior Transition Matrix (PRTM). Then it uses PRTM to generate a position status matrix.

#### 1) Prior Transition Matrix (PRTM)

An entry in PRTEM, for example, PRTM[A][C], indicates the relationship of Event_A with Event_C. It has three elements: Prior Count (PC), Ready Switch (RS) and Occurs in Same sequence Count (OSC). Note PRTM[A][C] is not the same as PRTM[C][A].

TSM creates a PRTM based on a list of the event of interest provided by users and initializes all events' PC and OSC value to 0 and RS to "Not ready".

When scanning a sequence, TSM performs the following operations for each event, denoted by event_n.

1. Set PRTM[n][k][RS] to "Ready". Here "k" refers to another event, event_k.
2. Look up other event_k in PRTM. If PRTM[k][n][RS] is "Ready", then add "1" to PRTM[k][n][PC] and change PRTM[k][n][RS] to "Not ready";
3. Use a counter to count how many times event_n appears in the sequence.

After scanning the sequence, TSM updates events' OSC in PRTM with events' occurrence counts. For example, if Event_A occurs once in Sequence_1 and Event_C occurs twice in Sequence_1, then after scanning sequence_1, TSM adds 1 to PRTM[A][C][OSC] and adds 2 to PRTM[C][A][OSC].

The pseudo code for generating PRTM is as follow.

```
def PRTM_gen (PRTM, event_sequence):
    for event in interested_event_list:
        #occur counter initialize
        event_count_list[event] = 0
    for event_n in event_sequence:
        event_count_list[event_n] += 1
        for event_k in interested_event_list:
            PRTM[event_n][event_k][RS] = "ready"
            if PRTM[event_k][event_n][RS] == "ready"
                PRTM[event_k][event_n][PC] += 1
                PRTM[event_k][event_n][RS] = "Not ready":
    #Scanning event_sequence finished
    for event_n and event_k in interested_event_list:
        if both event_n and event_k occurred in event_sequence:
            PRTM[event_n][event_k][OSC] += event_count_list[event_n]
            PRTM[event_k][event_n][OSC] +=event_count_list[event_k]
    return PRTM
```

#### 2) Generating position status matrix(POSM) from PRTM

Table III shows the PRTM created for Example 2 (Table II).

TABLE III. PRTM generated from Event Sequence Example

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | | [1, 3] | [2, 2] | [3, 3] | [3, 3] |
| B | [2, 3] | | [3, 3] | [4, 4] | [4, 4] |
| C | [0, 4] | [0, 6] | | [3, 6] | [3, 6] |
| D | [0, 3] | [0, 4] | [3, 3] | | [4, 4] |
| E | [0, 3] | [0, 4] | [3, 3] | [0, 4] | |

Each entry in the PRTM contains [PC, OSC]. To analyze the position relation of two events (say event_n and event_k) position, we need to define a support count. A support count is assigned by the lower OSC of PRTM[n][k] and PRTM[k][n] and is used as a threshold to tolerate noise. Then we use PRTM[n][k][PC] and PRTM[k][n][PC] to compare the support counts to understand their positions.

1. If PRTM[n][k][PC] is greater than or equal to the support count but PRTM[k][n][PC] is less than the support count, then event_n is prior to event_k and event_k is after event_n.
2. If both PRTM[n][k][PC] and PRTM[k][n][PC] are greater than or equal to the support count, then event_n and event_k are prior to each other. If event_n's OSC is greater than event_k's, then event_n wraps event_k.
3. If both PRTM[n][k][PC] and PRTM[k][n][PC] are greater than or equal to the support count and PRTM[n][k][OSC] and PRTM[k][n][OSC] are equal, then the one with a greater PC is prior to the other event.

4. If the PCs of both event_n and event_k are less than the support count, then they are tangling.

Now consider the example in TABLE III. For event_A and event_B, their support count is 3. However, PRTM[A][B][PC] = 1 and PRTM[B][A][PC] = 2, i.e., both of them are less than the support count. So event_A and event_B are tangling. For event_A and event_C, their support count is 2 (PRTM[A][C][OSC]), PRTM[A][C][PC] = 2, however, PRTM[C][A][PC] = 0. Then event_A is prior to event_C and event_C is after event_A. For event_C and event_D, their support count is 3 (PRTM[D][C][OSC]), and PRTM[C][D][PC] = PRTM[D][C][PC] = 3. Thus, wrapping is the position relation between event_C and event_D. As PRTM[C][D][OSC] = 6, which is greater than PRTM[D][C][OSC], event_C wraps event_D (or event_D is wrapped by event_C).

Users can assign two thresholds to tolerate noise:

1. *Minimal occurrence count threshold* for identifying rare events. If PRTM[n][k][OSC] is less than the minimal occurrence count threshold, it means the occurrence count of event_n and event_k in the same sequence is very small.
2. *A support count ratio threshold*. For instance, assume the support count of event_n and event_k is 100 and the threshold as 90%. If we observe event_n is prior to event_k for 90 times, then we say event_n is always prior to event_k.

By applying the preceding rules, we build a position status matrix (POSM), as shown in Table IV.

TABLE IV. POSITION STATUS MATRIX (POSM)

| Event | Prior | After | Wrapping | Wrapped | Tangling |
|---|---|---|---|---|---|
| A | C, D, E | | | | B |
| B | C, D, E | | | | A |
| C | | A, B | D, E | | |
| D | E | A, B | | C | |
| E | | A, B, D | | C | |

*3) Generating event patterns from position status matrix by using topology*

Based on the position relation among events, we can build event patterns. Our *topology-aware event pattern building process* works as follows.

1. We use a Waiting Event List (WIL) to store all events and a Temporary Sequence List (TSL) to store all temporary event patterns.
2. In each round, we randomly select one Waiting Event (WI) from WIL and remove it from WIL.
3. If TSL is empty, we append length-1 sequence [WI] to TSL, then use this WI's Tangling Events (WITI) from this WI's tangle list as length-1 sequences [WITI], and add sequence [WITI] to TSL.
4. If TSL is not empty, we compare WI with all Temporary Sequences (TS) in TSL.
   a) If WI is already in the TS or WI tangles with some event in the TS, then continue.
   b) If WI can be inserted into the TS based on POSM, we add a new sequence [TS-WI] to TSL, and add the WI's Tangling Events (WITI) to the TS. If any WITI is successfully added to the TS, we add a new sequence [TS-WITI] to the TSL.
   c) WI cannot be added to any TS in TSL, add length-1 sequence [WI] and all length-1 sequences [WITI] to TSL.
5. Repeat Steps 2 to 4 until WIL is empty.
6. If any TS in TSL has events with Wrapping/Wrapped position, we add one more wrapping event to the TS. The insertion position is after all events which have "Wrapping" or "After" position with it.

The TSL that is generated from POSM contains all of the long event patterns which events of interest are included.

The pseudo code of our proposed topology-aware sequence mining algorithm is as follows.

```
def TSL_gen (PRTM, POSM, WIL):
    TSL = {}
    while not WIL == []:
        WI = WIL.random_select ()
        WIL.remove (WI)
        if TSL == {}:
            TSL.append([WI])
            for WITI in POSM[WI]["tangle"]:
                TSL.append([WITI])
            continue
        WI_inserted = 0
            for TS in TSL:
                TS_WI = TS_insert(WI, TS, POSM)
                if not TS_WI == []:
                    WI_inserted = 1
                #WI can be inserted into TS
                    if not TS_WI in TSL:
                        TSL.append(TS_WI)
                    for WITI in POSM[WI]["tangle"]:
                        TS_WITI = TS_insert(WITI, TS, POSM)
                        if not TS_WITI in TSL:
                            TSL.append(TS_WI)
                    TSL.remove (TS)
            If WI_inserted == 0:
            #WI cannot be inserted into any TS
                TSL.append([WI])
                for WITI in POSM[WI]["tangle"]:
                    if not [WITI] in TSL:
                        TSL.append([WITI])
    return TSL
```

The topology-aware sequence mining algorithm scans events only once and does not generate a large number of temporary subsequences. Unlike FreeSpan or PrefixSpan, TSM uses tangling tags to determine if two events can be placed in the same sequence or not before generating any Temporary Sequence (TS). If a Waiting Event (WI) can be added to a Temporary Sequence, all Waiting Tangling Events (WITI) cannot be added to the sequence generated from TS and WI ([TS_WI]). We also note it is possible that all WITIs may be added to TS. That is why we try to add WITI to TS and generate another sequence [TS-WITI]. In this way, TSM can reduce a large number of subsequence comparison and merging operations.

TSM performs some matrix search and comparison operations. It needs to search WI's position status with events in the TS from Position Status Matrix (POSM) to find the

insertion position in the TS. When a new TS is added to TSL, TSM needs to compare this TS with existing TSes in TSL to see if this TS already exists. Such search and comparison operations do not happen very often and they are not computed intensive.

## V. GENERATING EVENT PATTERNS – CASE STUDY

In this section, we use an example to illustrate how TSM builds event patterns as described in Section IV. In the example shown in TABLE II, there are five events: A, B, C, D, and E. The Wait-Event List (WIL) includes these five events.

### *A. Using event order in WIL to process WIs*

**Step 1**: processing event A.
As TSL is {}, TSM adds pattern [A] and A's tangle event [B] to TSL. Thus, TSL = {[A], [B]}

**Step 2**: processing event B.
TSM tries to add B to each TS in TSL. As B tangles with A and pattern [B] already exists. TSM does not change TSL. Still TSL = {[A], [B]}

**Step 3**: processing event C.
TSM tries to add C to [A] and [B]. As C is after both A and B, C can be appended to [A] and [B] as [A, C] and [B, C]. TS [A] and [B] are then removed from TSL. So, TSL = {[A, C], [B, C]}

**Step 4**: processing event D.
D is after A and B, and D is wrapped by C. TSM discovers two new patterns [A, C, D], [B, C, D]. Then [A, C] and [B, C] are removed from TSL. Now TSL = {[A, C, D], [B, C, D]}

**Step 5**: processing event E.
E is after A, B, and D, and wrapped by C. TSM appends E to each TS in TSL. So, TSL = {[A, C, D, E], [B, C, D, E]}

### *B. Using user-defined order [C, A, E, B, D] to process WIs*

**Step 1**: processing event C.
C has no tangle event. TSM addes pattern [C] to TSL. Thus TSL = {[C]}

**Step 2**: processing event A.
A can be appended to pattern [C]. As A has a tangling event B, TSM appends B to pattern [C]. Then TSL = {[A, C], [B, C]}

**Step 3**: processing event E.
E is after A and B, and E is wrapped by C. TSM appends E to pattern [A, C] and [B, C], and removes [A, C] and [B, C] from TSL. So TSL = {[A, C, E], [B, C, E]}

**Step 4**: processing event B.
As B cannot be appended to TS [A, C, E] and B is already included in TS [B, C, E], TSM does not change TSL. Now TSL = {[A, C, E], [B, C, E]}

**Step 5**: processing event D.
As D can be appended to TS [A, C, E] and [B, C, E], TSM adds two new patterns TS_D: [A, C, D, E] and [B, C, D, E] and remove TS [A, C, E] and [B, C, E] from TSL. So, TSL = {[A, C, D, E], [B, C, D, E]}

From the preceding two cases, we can see both the generated TSL and the time complexity are independent of the order in which TSM processes events in WIL.

### *C. Processing and adding wrapping events to TS.*

As patterns [A, C, D, E] and [B, C, D, E] have event C and its wrapping events D and E in them. TSM adds event C to the two patterns. The insertion position is after D and E. Thus, TSL becomes {[A, C, D, E, C], [B, C, D, E, C]}.

### *D. Extracting subsequences from TSL to match event sequences*

As we discuss in Section III.C: the topology-aware event sequence building method, TSL contains all long event patterns. However, some patterns may not include events of interest. Thus, those patterns may not be able to match with event sequences.

In the preceding example, the set of events appear in event sequences are different, as shown in Table V.

TABLE V. EVENT SETS OF EVENT SEQUENCES

| Sequence_id | Event Sequence | Event Set |
|---|---|---|
| Seq_1 | [A, B, C, D, E, C] | A, B, C, D, E |
| Seq_2 | [B, A, C, D, E, C] | A, B, C, D, E |
| Seq_3 | [B, A, D, E] | A, B, D, E |
| Seq_4 | [B, C, D, E, C] | B, C, D, E |

In the table, we can see two event patterns in TSL cannot match with Sequence_3. On the other hand, we can use Sequence_3's event set {A, B, D, E} to extract two subsequences, that is [A, D, E] and [B, D, E]. Thus, the event pattern list becomes

{[A, C, D, E, C], [B, C, D, E, C], [A, D, E], [B, D, E]}.

## VI. EVENT PATTERN VERIFICATION

TSM can significantly improve the efficiency of event pattern mining and reduce computation overhead. All events are important. TSM randomly selects one WI from WIL to process each time, which makes some WIs processed early while other WIs are processed late.

To make sure the event patterns that TSM produces are correct. We verify the following three requirements.

- **R1.** All event patterns are fully generated. In other words, no more events can be added to any event patterns.
- **R2.** Event patterns are independent of the order in which WIs are processed.
- **R3.** Event patterns should not conflict with the sequences that are learned. If one sequence does not contain all events that an event pattern possesses, then a subsequence of the event pattern must be capable of matching the sequence.

Correspondingly, we develop three functions to verify the preceding requirements.

- **F1.** Try to add all events from WIL to all TS in TSL. If any WI from WIL can be added to any TS, then the TS is not fully generated.

**F2.** Manually assign different orders for processing WIs and apply these orders to build TSL. Then compare every two TSLs. If one TSL has some event patterns that the other TSL does not have, then event patterns are affected by the processing order.

**F3.** Use event patterns in TSL to map event sequences. Scan an event pattern and subsequences extracted from it and record how many times one event pattern and its subsequence can be matched from the first event to the last one.

Each event pattern has one event which has a minimal occurrence count in sequences than other events in this pattern. If this event pattern's successful occurrence count is no less than the minimal event's occurrence count, we conclude this event pattern is correct.

## VII. Performance Evaluation

### A. Experiment setting and test cases

We have implemented a prototype software of TSM and conducted experiments using Mutrino's system logs. The Mutrino system [10], sited at Sandia National Laboratories, is a Cray XC40 system with 118 nodes. It is a test environment of the Trinity production supercomputer, the 6th most powerful supercomputer in the world. The dataset that we use contains all syslogs collected from 2/11/2015 to 6/13/2016 on Mutrino.

We apply our recently developed System Log Event Block Detection (SLEBD) tool [18] to extract event blocks from groups of log messages, and convert the raw log messages to event block lists. We randomly select 18 nodes' event lists for the same period of time (two days). Among the 18 event list files, the shortest one contains 545 events and the longest one has 970 events. Table VI lists the number of events and events of interest in our test cases.

TABLE VI. Test Case Statistics

| | |
|---|---|
| Number of system events | 13156 |
| Number of interested common event types | 132 |
| Number of interesting events from test cases | 8981 |
| Shortest event list (in events) | 545 |
| Longest event list (in events) | 970 |

We run experiments on computer servers, each of which is equipped with Intel Xeon X3460 (8 cores, 2.8GHz) and 32 GB DRAM, and runs CentOS 7.3.1611.

### B. Verification results

We run TSM on the 18 event lists and analyze 8981 events of interest. TSM builds 364 event patterns. The longest pattern contains 84 events and the shortest pattern includes 75 events.

We test the three requirements as described in Section VI to verify the correctness of our experimental results. We find that no event of interest can be added to any of the 364 event patterns, all of the 364 patterns match with our 18 event lists, and their matching counts are equal to or greater than the minimal-occurrence event's occurrence count. The event pattern that has the smallest matching count succeeds 33 times. We run TSM 10 times and a random event processing order is used each time. The 10 runs produce the same (364) event patterns.

### C. Performance results

Factors, such as event list size and the number of events of interest, may influence the performance of TSM, specifically, the building speed of event patterns. We conduct a set of experiments to evaluate the influence. We randomly select 100 events of interests out of the 132 common events. For comparison, we also two event list sets, one of which contains eight lists randomly selected from the 18 lists, and other set contain the remaining lists.

All TSL event patterns generated from the experiments are tested and they all pass the verification test as described in Section VI. Table VII shows the time that TSM uses to create PRTM, POSM and build event patterns.

TABLE VII. Execution Time of TSM

| Seq. size | Event of interest | Event count | Seq. analysis and PRTM gen time (s) | POSM gen time (s) | TSL build time (s) | TSL size |
|---|---|---|---|---|---|---|
| 18 | 132 | 8981 | 1.11 | 0.165 | 2.28 | 364 |
| 18 | 100 | 7031 | 0.68 | 0.08 | 0.29 | 33 |
| 5 | 132 | 2127 | 0.34 | 0.16 | 1.85 | 378 |
| 10 | 132 | 4961 | 0.66 | 0.16 | 1.67 | 296 |

For ease of our discussion, we use "F_*i*_I_*j*" to denote the set of an experiment with (i) event sequences and (j) events of interest. For example, F_18_I_132 processes 18 event sequences and 132 events.

From Table VII, we find the POSM generation time of F_18_I_132, F_5_I_132 and F_10_I_132 is around 0.16 second. F_18_I_100 processes only 100 events of interest, but its POSM generation time is 0.08 second. Additionally, the TSL build time in the three experiments (F_18_I_132, F_5_I_132, and F_10_I_132) are close to 2 seconds and their sizes of TSL are close. The TSL build time of F_18_I_100 only takes 0.29 second. These findings show TSM's execution time is influenced more by the number of events of interest than the sequence length.

We also notice that F_18_I_100 only produces 33 event patterns. After further analysis, we find that there are 57 events among the 100 event types whose tangling lists are empty. Among the 132 event types, 67 events have empty tangling list. To further understand the relationship between event patterns and events' tangling lists, we conduct the following experiment.

TABLE VIII. Execution Time for New 18 Sequences and 100 Events

| Seq. size | Event of interest | Event count | Seq. analysis & PRTM gen time (s) | POSM gen time (s) | TSL build time (s) | TSL size |
|---|---|---|---|---|---|---|
| 18 | 100 | 7543 | 1.13 | 0.160 | 0.92 | 364 |

After TSM generates POSM from the 132 events, we create a new event list which has 65 (i.e., 132 - 67) events whose tangling lists are not empty. We randomly select 35 events whose tangling lists are empty to create a 100-event list and build event patterns. Table VIII presents the results. In the table, we find the sequence analysis time, PRTM generation

time, and POSM generation time are the same as those for F_18_I_132. This is because TSM uses the same event set to create POSM. Then the 100-event list is selected from POSM. We also find the size of TSL built from the 100-event list is the same as that in F_18_I_132. Each event pattern in F_18_I_100's TSL matches with a corresponding longer event pattern in F_18_I_132's TSL and the former pattern is a subsequence of the latter pattern.

If TSM generates event patterns among events with an empty tangling list, only one pattern is produced. We call it common sequence. The difference between F_18_I_132 and F_18_I_100 is that F_18_I_132's common sequences are generated by 67 events with an empty tangling list, while F_18_I_100's common sequences are generated by 35 events. F_18_I_100's common sequences are subsequences of that of F_18_I_132. If TSM builds event patterns among the 65 events with non-empty tangling lists, 364 subsequences are generated, which is the case for F_18_I_132 and F_18_I_100. The TSLs generated from F_18_I_132 and the F_18_I_100 contain event patterns merged from the 364 subsequences and the common sequences. This explains why an event pattern from F_18_I_100 matches with a corresponding longer pattern from F_18_I_132. Thus the reason that only 33 event patterns are built in F_18_I_100 is that the randomly selected 43 events with non-empty tangling lists can only generate 33 subsequences patterns.

We also note that even though F_18_I_132 and F_18_I_100 have the same set of events with non-empty tangling lists, the time for building their TSLs varies much. This is because the TSL build time is influenced by the size of an event set. Adding events with an empty tangling list still takes time.

### *D. Performance comparison with FreeSpan and PrefixSpan*

We download two PrefixSpan programs in Python from Github [11, 12]. We also implement TSM in Python. We run the three programs under F_18_I_132. The two downloaded programs crash before completion.

To get some useful results from the programs, we extract smaller test cases to evaluate the performance of PrefixSpan and TSM. We randomly select 3 event sequences from the 18 sequences and extract 5, 10, 15, and 20-25 events from the beginning to create new test sets. Figure 2 shows the execution time of the three programs, i.e., two downloaded PrefixSpan programs and our TSM implementation.

From Figure 2, we can see PrefixSpan1 takes 200 seconds when the size of the test set is 21, and PrefixSpan2 takes 181 seconds for 25 events. In contrast, TSM's execution time is always less than 0.01 second. The figure shows the size of the event set increases, the execution time of PrefixSpan increases sharply, making PrefixSpan unsuitable for processing the overwhelming number of events on production HPC systems.

PrefixSpan1 and PrefixSpan2 extract 751615 length-1 to length-21 subsequences. Then we use a subsequence filtering program to remove subsequences that are too short or do not include untangling events of interest. Two long event patterns are produced, one of which is length-12 and another is length-13. TSM generates the same event patterns. Therefore, TSM is functionally correct but much more efficient than PrefixSpan.

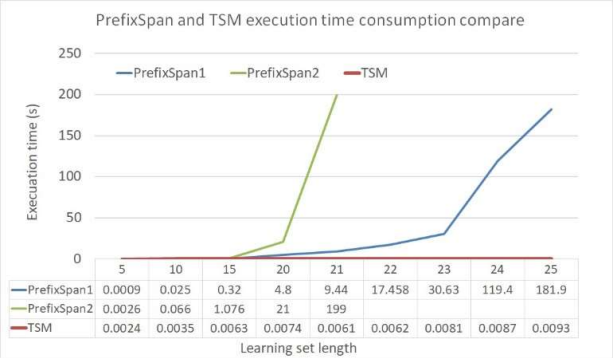


Figure 2. The execution time of PrefixSpan and TSM.

As the two downloaded PrefixSpan programs crash even for a small event set, we implement PrefixSpan in Python by ourselves. We run our PrefixSpan code for F_18_I_132 and it takes 1.34 seconds to generate all length-2 subsequences and 245 seconds to generate all length-3 subsequences. We ran the program for 15 minutes hoping to get length-4 subsequences. However, the results did not show up and so we terminated the execution.

## VIII. Deep Learning-Based Event Sequence Classification and Anomaly Detection

Deep learning algorithms, such as Recurrent Neural Networks (RNN) [13] and Long Short-Term Memory (LSTM) [14], can help us classify event sequences.

In our recent work [18], we developed a system log event block detection (SLEBD) method that consolidates log messages that occur together into event blocks and converts raw system logs into event block lists. This enables us to detect the event-level system and component anomalies. If a log message which should belong to an event block is found outside that event block in the produced event list, an anomaly related to that event block is detected.

On Mutrino [10], we detect an event block Block_10 occurs 932 times normally and 39 times with anomalies from a 300-day log. We extract 500 events prior to each normal occurrence or anomaly of Block_10. These event sequences are grouped into two clusters: successful and failed. The two clusters are biased, as the successful set is much larger than the failed set. We use the SMOTE method [15] to balance the two sets. We then divide the produced dataset into a training set (which contains 2/3 of the event sequences) and a test set (which contains the rest 1/3 of event sequences). We build a Recurrent Neural Network, using a deep learning library Keras [16] to generate an LSTM model from the training set. The LSTM model is evaluated by using the test set. The experimental results from three-fold cross-validation show a promising classification performance, i.e., the classification accuracy for the successful cluster and the failed cluster is 95% and 86%, respectively.

In HPC systems, log messages are streamed to the log service. It is unknown what messages or system events will appear in the future. To address this issue, we propose to use TSM to detect event patterns from a trained LSTM model to predict future events. We aim to discover events in the *critical paths* that can be used to describe system behaviors. Deviations from the critical paths indicate changes of a system behavior or appearances of anomalies.

TABLE IX. TRAINING SEQUENCES FOR LSTM

| LSTM TRAINING SEQUENCES | |
|---|---|
| SEQ_1 | E0, **E1**, E2, **E3**, **E5**, E6, **E7** |
| SEQ_2 | **E1**, E0, E2, **E3**, E4, **E5**, **E7** |
| …… | …… |
| SEQ_N | E0, **E1**, E2, **E3**, **E5**, E4, **E7** |

Table IX shows an example with a set of training sequences for building an LSTM model for an event, say E8. These events happen prior to E8 in a window. The event sequences are not the same. We use TSM to discover a critical path from these sequences as [E1, E3, E5, E7]. We run an event monitor on an HPC system. If the critical path [E_1, E_3, E_5, E_7] is detected, TSM can predict E8 will happen. Then TSM uses the trained LSTM model to check event sequences containing the critical path and predicts if E8 has an anomaly or not.

In our experiments, we leverage SLEBD [18] to convert 300-days system logs collected from the Mutrino supercomputer to an event block list. Then we select 10 event blocks and extract sequences containing 500 events prior to each of the 10 event blocks. LSTM models are trained to detect the critical paths, which are then used to predict system behavior (execution) and anomalies for the following 30 days on Mutrino. Our prediction accuracy is above 90%.

## IX. CONCLUSIONS

Log analysis for system behavior characterization in HPC systems has continuously been an important research topic. In this paper, we present a novel topology-aware sequence mining method (TSM) to discover critical event patterns in which events of interest are included. TSM scans event sequences once and is capable of produce long event patterns in a cost-effective manner.

To assure the correctness of the discovered event patterns, we present verification requirements and functions to check event patterns. In a case study, we use system logs collected from a production supercomputer and build event patterns from its event sequences. Our verification tests prove that the generated event patterns meet the requirements and are correct.

We further explore TSM to identify those event patterns that can describe system behaviors (critical paths). We leverage RNN, more specifically LSTM, to accurately detect anomalous event sequences and predict the occurrence of events in critical paths.

To the best of our knowledge, TSM is the first of its kind that uses topology information to discover event patterns. It can generate long event patterns with a low computation overhead and outperform the existing sequence mining methods, which makes TSM suitable for online event analysis and anomaly detection on production HPC systems.

## ACKNOWLEDGEMENTS

The Mutrino Dataset is released by the Holistic Measurement Driven Resilience (HMDR) project to the research community in support of Extreme-Scale HPC Resilience Research. HMDR is funded by the U.S. Department of Energy Office of Advanced Scientific Computing Research.

This work was funded in part by a LANL grant and an NSF grant CCF-1563750. Los Alamos National Laboratory is supported by the U.S. Department of Energy contract DE-AC52-06NA25396.

## REFERENCES

[1] R. Srikant and R. Agrawal. Mining quantitative association rules in a large relational table. In Proc. of ACM International Conference on Management of Data (SIGMOD), 1996.

[2] Hu, M., Zheng, G., Wang, H.. Improvement and research on Aprioriall algorithm of sequential patterns mining. In Proceedings of International Conference on Information Management, Innovation Management and Industrial Engineering, 2013.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In Proc. 1994 Int. Conf Very Large Data Based (VLDB), 1994.

[4] Mohammed J. Zaki, SPADE: An Efficient Algorithm for Mining Frequent Sequences, Machine Learning, v.42 n.1-2, p.31-60, 2001.

[5] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu. Freespan: Frequent pattern-projected sequential pattern mining. In Proc. Int. Conf. Knowledge Discovery and Data Mining (KDD), 2000.

[6] Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U., and Hsu, M.-C. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In Proc. of International Conference on Data Engineering (ICDE), 2001.

[7] Wang L (2013) Directed acyclic graph. Encyclopedia of Systems Biology: 574-574

[8] Hazewinkel, Michiel, Topology general, Encyclopedia of Mathematics, Kluwer Academic Publishers, 2001.

[9] George, A.; Binu, D. (2013). An Approach to Products Placement in Supermarkets Using PrefixSpan Algorithm. Journal of King Saud University-Computer and Information Sciences.

[10] Dataset downloaded from http://portal.nersc.gov/project/m888/resilience/

[11] https://github.com/rangeonnicolas/PrefixSpan/blob/master/PrefixSpan.py

[12] https://github.com/chuanconggao/PrefixSpanpy/blob/master/prefixspan.py

[13] Mikolov T, Kombrink S, Burget L, et al. Extensions of recurrent neural network language model[C]//Acoustics, Speech and Signal Processing (ICASSP), IEEE International Conference on. IEEE, 2011.

[14] Hochreiter, S. & Schmidhuber, J. Long short-term memory. Neural Comput. 9, 1735–1780 (1997).

[15] Jeatrakul P., Wong K.W., Fung C.C., Classification of Imbalanced Data by Combining the Complementary Neural Network and SMOTE Algorithm. In Neural Information Processing. 2010.

[16] https://keras.io/.

[17] Rakesh Agrawal and Ramakrishnan Srikant Fast algorithms for mining association rules. Proceedings of the 20th International Conference on Very Large Data Bases (VLDB), 1994.

[18] Zongze Li, Matthew Davidson, Song Fu, Sean Blanchard, Michael Lang. Event Block Identification and Analysis for Effective Anomaly Detection to Build Reliable HPC Systems. In *Proceedings of the 20th IEEE International Conference on High Performance Computing and Communications (HPCC)*, 2018.