A Visual Analytics Framework for the Detection of Anomalous Call Stack Trees in High Performance Computing Applications

Cong Xie, Wei Xu, *Member, IEEE* and Klaus Mueller, *Senior Member, IEEE*

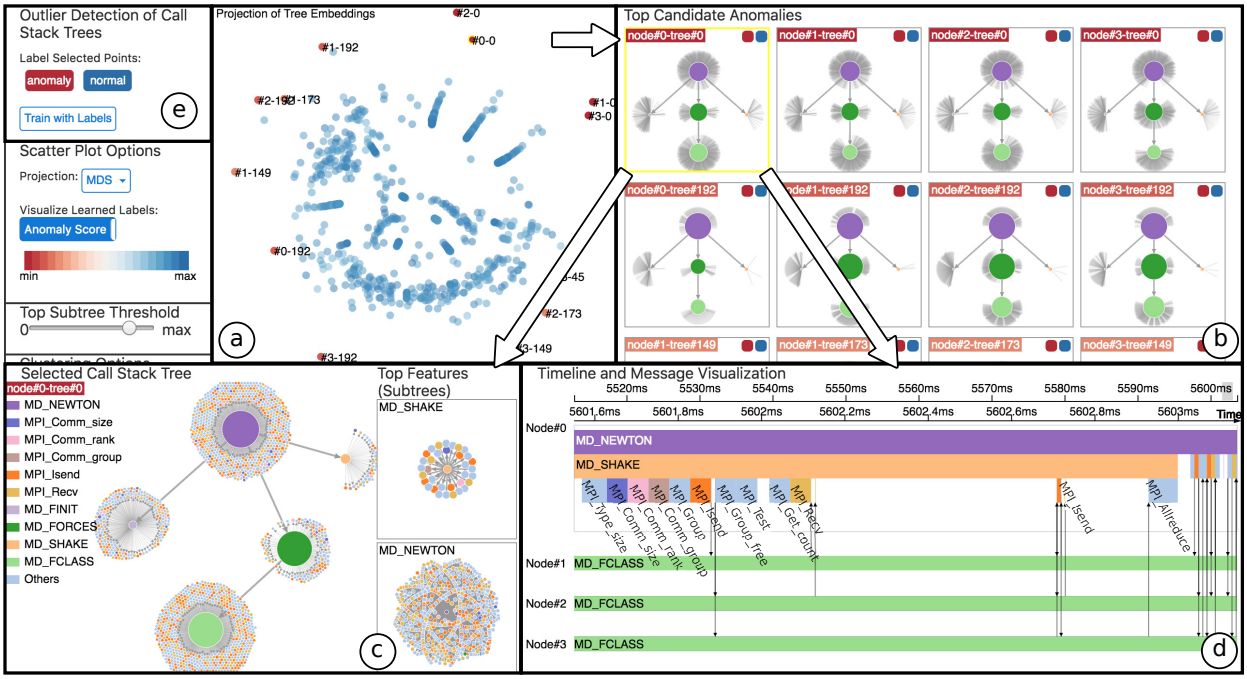


Fig. 1. The interface of our system for anomalous call stack tree (CSTree) detection. (a) The scatter plot shows the projection of our stack2vec embeddings of the CSTrees. Each point in the projection represents a CSTree. (b) Summary structures of the top candidate anomalies from (a). (c) The user can investigate the detailed structure and the anomalous subtrees of a CSTree of interest. (d) The level-of-detail timeline visualization of the selected CSTree shows the temporal pattern of the invocations and the communications between the HPC nodes. (e) The user is able to label the CSTrees of interest after exploration to update the anomaly detection model.

Abstract—Anomalous runtime behavior detection is one of the most important tasks for performance diagnosis in High Performance Computing (HPC). Most of the existing methods find anomalous executions based on the properties of individual functions, such as execution time. However, it is insufficient to identify abnormal behavior without taking into account the context of the executions, such as the invocations of children functions and the communications with other HPC nodes. We improve upon the existing anomaly detection approaches by utilizing the call stack structures of the executions, which record rich temporal and contextual information. With our call stack tree (CSTree) representation of the executions, we formulate the anomaly detection problem as finding anomalous tree structures in a call stack forest. The CSTrees are converted to vector representations using our proposed stack2vec embedding. Structural and temporal visualizations of CSTrees are provided to support users in the identification and verification of the anomalies during an active anomaly detection process. Three case studies of real-world HPC applications demonstrate the capabilities of our approach.

Index Terms—Call Stack, Performance Visualization, Representation Learning, Active Learning, Anomaly Detection

1 Introduction

High Performance Computing (HPC) is imperative in many scientific domains. However, the supercomputer resources in use are limited and costly, thus adverse runtime behaviors and latencies can have a large negative impact. For this reason the detection of anomalies [8] in the parallel program execution is a critical mission. Anomalous function executions are usually identified from the detailed trace events of the HPC cluster. Trace events are sequences of all the activities of function entry, function exit, and message passing in an HPC node during application execution. Fig. 2 (a) shows an example sequence of trace events inside one execution of the `compute` function [42] [41] in an HPC node.

During their analyses, domain scientists, such as physicists solving complex molecular equations, typically detect the anomalous function executions based on their individual properties, such as execution time and exit timestamps. For example, a very large execution time of an `MPI_Wait` function may indicate that there is some unexpected communication delay between cluster nodes. That long execution of `MPI_Wait` is then identified as a candidate anomaly. However, using only the execution properties of each function independently typically fails to identify the real anomaly. Here are several scenarios where this

- *Cong Xie and Klaus Mueller are with Department of Computer Science at Stony Brook University. E-mail: {coxie, mueller}@cs.stonybrook.edu.*
- *Wei Xu is with Computational Science Initiative at Brookhaven National Laboratory. E-mail: {xuw}@bnl.gov.*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

problem occurs:

- The behavior of a function execution is affected by the children functions. For example, a `forward_comm_pair` with many `MPI_Wait` children is supposed to use more time than a `forward_comm_pair` with only one `MPI_Wait` child. In this case, the long execution time of `forward_comm_pair` may not be related to a delay of communication.
- Different parents of the same function may lead to different runtime behavior. For example, a `MPI_Wait` may have a longer duration when it is called by function `A` than function `B`. Nevertheless, a long `MPI_Wait` execution with parent `A` can still be a normal function and do not represent an anomaly.
- The latency of an execution in an HPC node may be related to the context nodes which it communicates with. For example, a function waiting for the response of a very busy HPC node will have a long delay.

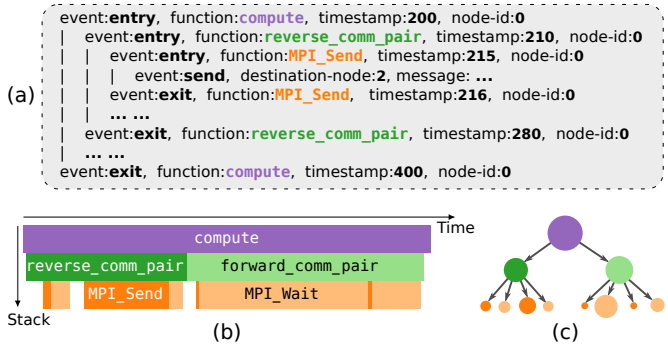


Fig. 2. (a) Example trace events of a `compute` execution in an HPC node. (b) The call stack of the execution can be reconstructed from the trace events. (c) The call stack can be represented as a directed tree with vertex weights.

As the cases given above illustrate, it is insufficient to define anomalous function execution using only the temporal information of the execution without the context. Conversely, the call stack (Fig. 2 (b)) during a function execution records both temporal and call path information of the function execution. Our approach takes advantage of this call stack information and models the comprehensive runtime behavior of an execution using a call stack tree (CSTree) representation (Fig. 2 (c)). Therefore, the anomalous behavior detection can then be formulated as the problem of finding anomalous tree structures in a call stack forest. Take the `compute` function in Fig. 2 as an example, our algorithm will return the `compute` invocations whose CSTrees are the most unexpected among all `compute` executions in the HPC cluster.

We propose a visual analytics approach for the detection of the anomalous trees with the user's domain knowledge. First, for the representation and comparison of the CSTrees, we propose stack2vec to convert each CSTree into a feature vector. Second, with this embedding vector in hand, we employ a One-Class Support Vector Machine (OCSVM) [43] to detect the top candidate anomalies. Finally, to aid a detailed investigation of a candidate, we devise dedicated visualizations of subtrees and timelines in order to show the top subtree features and temporal invocations, respectively. After verification, the user is then able to label the candidates as either normal or abnormal. The labeling information is fed back to refine the OCSVM model.

The main contributions of our paper are:

- We formulate the problem of finding anomalous runtime behavior as the detection of anomalous tree structure in a call stack forest, which improves upon the existing performance analysis by utilizing the context of function executions.
- We propose stack2vec, which optimizes the graph kernel approach and uses neural networks for the learning of tree representations in a forest.
- We propose a new visual anomaly detection approach with active learning strategy for finding, verifying and labeling the candidate anomalous CSTrees interactively.

The remainder of our paper is structured as follows. Section 2 reviews related work. Section 3 defines the problem and gives an overview of our approach. Section 4 introduces our algorithm for vector representation learning of CSTrees. Section 5 describes our visual analytics approach for anomalous CSTree detection with active labeling. Three case studies are used to validate our approach in Section 6. Section 7 ends with conclusions and future work.

2 Related Work

Performance analysis for heavy computation applications usually focuses on the data (e.g., trace events) collected by the instrumentation and measurement tools [2] [16] [29] [46]. Many techniques [9] [51] [52] [10] [11] [12] have been proposed for the visualization, analysis, and diagnosis of these execution data [13]. Our paper mainly focuses on the anomalous runtime behavior detection, which is one of the most important tasks in performance analysis [22].

2.1 Anomalous Behavior Detection in HPC

The existing anomaly detection approaches deal with different kinds of runtime behaviors that occur in HPC clusters, such as high I/O latency [48], large memory utilization [19] [49], and deadlock [3].

One challenge for the anomaly detection is that most approaches calculate an anomaly score based on the features of an execution, such as time [23] or memory usage [19] [49], while the contextual information (e.g., the structure of the program execution) is largely ignored. In contrast, our approach uses the context obtained from the call stacks to identify the potential anomalous executions.

Another problem is that the unsupervised learning algorithms detect outliers purely based on density [7] or clusters [20] in the dataset. However, sometimes a rare execution pattern does not necessarily indicate abnormal behavior. For example, an initialization subroutine is only invoked at the beginning of the program, but it is normal.

One solution to the second problem is reducing anomaly detection to a classification problem [17] [15] [5], which is able to utilize the labeling information provided by the user. However, the labeled data is usually not sufficient due to the complexity of human-based labeling. Active learning [45] [1] is a solution that reduces the human effort by choosing maximally informative samples for labeling. Our approach integrates the active learning [45] [21] strategy with One-Class SVM [43], which returns the top anomaly candidates. The user is then asked to inspect, verify, and provide labels for the most critical data instances to refine the model.

2.2 Performance Visualization in HPC

Visualization of the program execution structure [13] [22] is widely used for performance navigation and monitoring tasks [35] [53]. Jumpshot [55] and Vampir [28] show the call relations of functions in the trace events with a timeline. Profiling call tree visualizations [32] [38] seek to provide insight about the structure and runtime costs of the function calls. Other visualizations [13] [22] focus on executions and message communications, such as SyncTrace [24]. While these methods are effective for a human scientist to understand the execution context, it tends to be very time-consuming for the user to search and identify the anomalies manually among the typically large numbers of trace events. To bridge the gap between automatic anomaly detection and call tree visualization, we support a visually-guided training of the anomaly detection model in our structural and temporal visualizations.

2.3 Graph Structure Embedding

The major challenge of anomaly detection in a forest of trees lies in how to represent and compare the structural features. Recent research on neural graph learning provides an efficient solution for this problem. Node representations such as DeepWalk [40] and node2vec [18] focus on the embedding of a vertex in a single graph. These techniques, however, are unable to represent the structure of a graph. Convolutional neural networks (CNNs) [27] [39] learn the local structure of graph

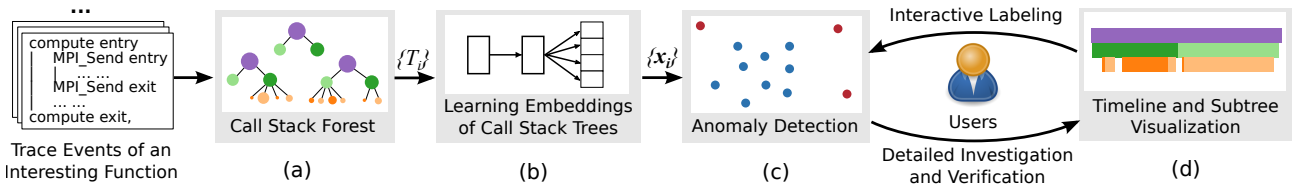


Fig. 3. Overview of our approach for anomalous execution detection. (a) The CSTrees are generated from the trace events. (b) Feature vectors are constructed for the CSTrees using stack2vec. (c) The candidate anomalous CSTrees are detected in the forest. (d) The user can investigate the candidates in detailed visualization. Labeling information provided by the user will be fed back to update the anomaly detection model.

nodes using an analogy to the pixel-level neighborhood of images. However, the labeling of the graphs for the training of the CNNs is usually not available in unsupervised anomaly detection.

Another family of techniques for graph isomorphism and similarity analysis is graph kernels, which decompose a graph into substructure features (e.g., random walk [25] and shortest path [6]). For example, Weisfeiler-Lehman graph kernels [47], extracting non-linear subgraphs, is one of the most popular approaches in practice [54]. Graph kernels sometimes lead to very high dimensional, sparse and non-smooth representations and thus yield poor generalizations [54]. Subgraph2vec [36] and graph2vec [37] deal with this problem by reducing the substructure features of graph kernels using deep learning approaches [31].

To learn the structural features of trees, our stack2vec follows the framework of graph2vec and optimizes the graph kernel computation for our CSTree representation.

3 Problem Definition and Approach Overview

3.1 Anomalous Execution Behavior and Call Stack Tree

Given a critical function `A` specified by the domain scientists and all of its invoked executions in an HPC cluster, we want to determine which invocations of `A` indicate anomalous runtime behavior. Here `A` is usually specified as the `main` function in each HPC node or a key function (e.g., `compute` [42] [41]) frequently invoked during the HPC application. From discussions with domain scientists we learned that anomalous behavior is always associated with large execution time (e.g., computation or communication delay) or unusual call path structure (e.g., child functions called in a loop with large counts).

As mentioned in Section 1, the call stack describes the contextual information of function execution. We represent the structure of an execution of `A` with a call stack tree (CSTree) $T = (V, E, \boldsymbol{w})$, which is a directed tree with weighted vertices rooted at `A` (Fig. 2 (c)). A vertex $v \in V$ represents a function invoked in the call stack and a directed edge $e \in E$ shows the call from a parent function to a child function. The vertex weight $w(v)$ is defined as the execution time of its function. With this representation, the anomalous behavior mentioned above can be implied from the structure of the CSTree. For example, a large vertex in a CSTree stands for long execution time of that function. And a vertex with many children of the same function may indicate that the parent function invokes the child function for multiple times in a loop.

By using all invoked executions of function `A` in the HPC cluster, a call stack forest can be generated from all its CSTrees. Our problem is then defined as follows: given a forest of CSTrees $\mathscr{T} = \{T_i\}$ of `A`, learn the anomaly labeling $\boldsymbol{l}$ of the CSTrees where $l_i = 1$ is normal and $l_i = -1$ is abnormal. The notations are summarized in Table 1.

Notation	Description
$T_i = (V, E, \boldsymbol{w})$	A CSTree of a function execution.
$v \in V$	A vertex, which is a function called during the execution of the root function.
$e \in E$	A directed edge, which is a call from a parent function to a child function.
S_i	The bag-of-subtrees of T_i.
$s \in S_i$	A subtree structure of T_i.
$\boldsymbol{x}_i$	The embedding vector of T_i.
$\mathscr{T} = \{T_i\}$	The forest of all CSTrees.
$\boldsymbol{l}$	The anomaly labeling vector of $\mathscr{T}$.

Table 1. Notations used in our paper.

3.2 Visual Analysis Tasks

Allowing experts to inject domain knowledge can greatly enhance automatic learning algorithms. A good way to do this is via visual interactions. After discussions with our domain users, we identified the following four tasks helpful in this process: **T1:** Gain an overview of the anomaly distribution. **T2:** Order the CSTrees to focus on the top anomalies. **T3:** Examine the detailed invocation structure of a CSTree of interest. **T4:** Examine the temporal patterns of a CSTree, including the message passings and execution durations.

3.3 Approach Overview

Based on these four essential tasks, our visual analytics approach detects anomalous CSTrees using the following four steps (Fig. 3):

Step 1 Generating the call stack forest: Given the trace events measured at the HPC nodes and a function of interest `A` specified by the user, a call stack forest $\mathscr{T} = \{T_i\}$ of `A` is constructed.

Step 2 Learning the tree representations: Each CSTree T_i is converted into a bag-of-subtree representation S_i using optimized Weisfeiler-Lehman graph kernels [47]. Based on the subtrees, the vector representations $\{\boldsymbol{x}_i\}$ for the CSTrees $\{T_i\}$ are learned by doc2vec [31]. The embeddings encode the structural similarities in a reduced space.

Step 3 Detecting anomalous CSTrees in the forest: With the vector representations of the CSTrees, a One-Class SVM is used to find the potential (candidate) anomalies after training with normal data.

Step 4 Visual investigation, verification, and labeling of the candidate anomalies: For the candidate anomalous CSTrees, the user can view their detailed structures, message communications, and top subtree features. Based on the observation, the user labels the candidates either as normal or abnormal based on available domain knowledge. The labels are then used to refine the One-Class SVM model.

Step 3 and Step 4 are performed iteratively until satisfying results are achieved. The user can also return to the previous steps for parameter tuning of the algorithm (e.g., embedding dimension in step 2).

4 Stack2vec: Learning Representations for Call Stack Trees

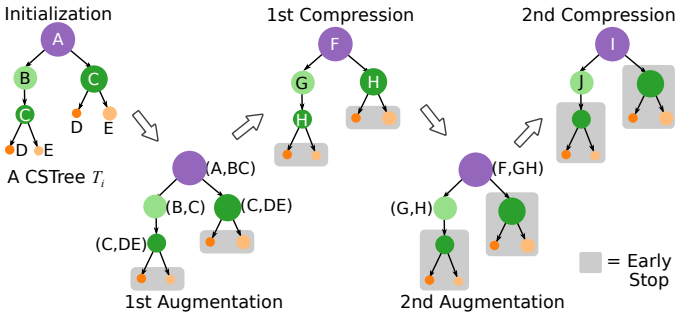
Our approach begins with the construction of the call stack forest $\mathscr{T} = \{T_i\}$ of the target function `A`. Each CSTree T_i is generated from the entry and exit activities occurring in an invocation of `A`. These activities are retrieved from the trace events. The message passings during the execution are also saved in the vertices of the sending and receiving functions. They can be provided as supplemental information during the visual analysis (Section 5.3.2). From our collaborating scientists we learn that they are less interested in the execution order of the children functions of `A` since this order is fixed by the source code design. As a result, the CSTree ignores the order of functions. Still, the user is able to view them in the timeline visualization (Section 5.3.2).

With the constructed call stack forest in hand, the challenge now is how to compute the structural similarities between different instances of CSTrees [44] for anomaly detection. In addition, most anomaly detection approaches require the input of fixed length feature vectors, rather than tree structures. As discussed in Section 2.3, graph2vec [37] is an effective solution which converts a tree structure to a feature vector. It combines two popular graph learning approaches: first representing the structural features of graphs by bag-of-substructures using Weisfeiler-Lehman Kernels [47], and then learning the graph embeddings using doc2vec [31]. In this work, we propose stack2vec that customizes these

two stages in graph2vec for the CSTree. Specifically, we optimize the original Weisfeiler-Lehman kernels for computing the bag-of-subtrees representation of our directed tree structures.

4.1 Accelerated Weisfeiler-Lehman Algorithm

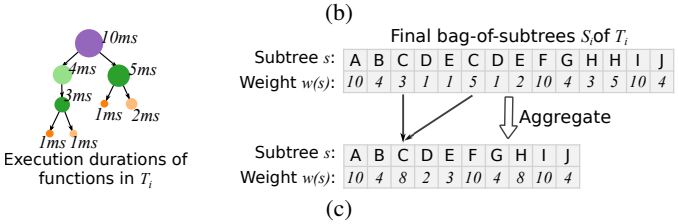
The Weisfeiler-Lehman (WL) algorithm [47] employs an iterative vertex relabeling process to extract substructures from graphs. In order to illustrate the WL algorithm, we use an example CSTree T_i rooted at function `A` in Fig. 4. At the beginning, the algorithm assigns an initial label string to each vertex in T_i. In our case, the vertex labels are initialized as their function names (see initialization in Fig. 4 (a)). In each iteration, the WL algorithm performs two operations on the vertex labels: augmentation and compression.



(a)

	Label assignments and their structures	New subtrees
Initialization	A= B= C= D= E=	A,B,C,D,E,C,D,E
1st Iteration	F=(A,BC)= G=(B,C)= H=(C,DE)=	F,G,H,H
2nd Iteration	I =(F,GH)= J =(G,H)=	I,J

(b)



(c)

Fig. 4. (a) An example of the WL relabeling algorithm for a CSTree T_i with 2 iterations. We optimize the original label updating process by early stopping, as shown in the gray backgrounded regions. (b) The table shows the i-degree subtree created in the ith iteration. (c) The final bag-of-subtrees contains all the generated subtrees. After aggregation, the weight of a subtree represents the total duration of its existence in T_i.

Label augmentation. The label of v is appended by the labels of its children, resulting in a signature string of v. For example, in the 1st augmentation of Fig. 4 (a), the signature `(A,BC)` of the root is generated by appending its children `B` and `C`.

The signature string of v represents the local structure of v. For example, the signature `(A,BC)` generated in the 1st iteration stands for a 1-degree subtree (i.e., subtree with depth of 1) of `A`. To be more general, the signature in the ith iteration represents an i-degree subtree structure (i.e., subtree with depth of i), since it has explored all descendants whose distances to v are less or equal to i. The detailed structure of the generated subtrees are shown in the middle column of Fig. 4 (b). As a result, the local structures of two vertices are identical if and only if their signatures are the same.

Label compression. A new signature incurred in this iteration is compressed. For example, a new label `F` is assigned to the signature `(A,BC)` in the first iteration in Fig. 4 (b). Therefore, the label `F` represents the 1-degree subtree structure. The meaning of relabeling a same vertex is to discover its subtrees of different degrees. For example, although `A`, `F`, `I` are all labels of the purple vertex, they stand for 0-degree, 1-degree, and 2-degree substructures rooted at that vertex, respectively (see the middle column in Fig. 4 (b)).

The two steps are repeated until a max iteration count set by the user is reached. By default, our algorithm finds the max depth d of the CSTrees in $\mathscr{T}$ and sets the max iteration count to $(d-1)$ to make sure all the subtree structures in the forest are explored. Finally, all the 0-degree (i.e., a vertex) to $(d-1)$-degree subtrees (see the last column in Fig. 4 (b)) will be produced for the bag-of-subtree representation (Section 4.2). The relabeling process is performed for all the trees in the forest to calculate their bag-of-subtrees.

To optimize the original WL algorithm for our directed tree structure, we introduce early stopping in the relabeling process (see the vertices in the grayed-out regions in Fig. 4 (a)). For a vertex v, if the maximum depth of the subtree rooted at v is h, we will stop updating v after the hth iteration. This is because the $(h+1)$th iteration will generate a $(h+1)$-degree subtree, which does not exists for v. Take a dark green vertex `C` in Fig. 4 (a) as an instance, there are only a 0-degree subtree `C` and a 1-degree subtree `(C,DE)` rooted at `C`. Our algorithm stops updating the label and signature of this vertex after the 1st iteration.

4.2 Bag-of-subtrees Representation

In stack2vec, all the subtrees (see the last column in Fig. 4 (b)) generated from the WL relabeling process are inserted into a multiset $S_i = \{s\}$ (see the first table in Fig. 4 (c)). Similar to bag-of-words, S_i is the bag-of-subtrees representation of a CSTree T_i. The subtree "corpus" of the forest $\mathscr{T}$ is the collection of all unique subtree structures in all the bag-of-subtrees $\{S_i\}$.

To encode the temporal information, we assign the weight $w(s)$ of a subtree $s \in S_i$ to be the weight $w(v)$ of the subtree root v (i.e., execution time of v). Essentially, the execution of a subtree root covers the executions of all of its descendant functions. Therefore $w(s)$ represents the existence duration of the substructure s in the call stack, as shown in the first table in Fig. 4 (c).

Since a substructure can occur multiple times in T_i, we count the "frequency" of a unique subtree structure in T_i by aggregating the subtrees with the same label in S_i. For example, the two subtrees of `C` are aggregated in the second table of Fig. 4 (c). The aggregated weight is the summed weight of the original subtrees. Equivalent to the word frequency in a bag-of-words, the weight of an aggregated subtree s represents the total duration of s in T_i.

4.3 Learning Tree Embeddings using Skip-Gram Model

The generated bag-of-subtrees can now be directly used as the input of the anomaly detection. However, as mentioned in Section 2.3, a large amount of subtree structures can make this representation very sparse. In analogy to the neural embedding of documents, stack2vec embeds the CSTrees in $\mathscr{T}$ with the same framework as doc2vec [31]. Each CSTree T_i is regarded a document and the subtrees $s \in S_i$ are the words, whose "frequencies" are the corresponding subtree weights $w(s)$.

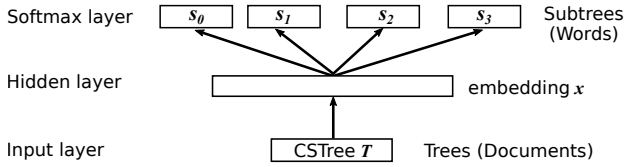


Fig. 5. A doc2vec network is adopted for the tree representation learning with an analogy to documents. This model predicts the subtree occurrences in a CSTree. The outputs of the hidden layer are used as the embedding vectors of CSTrees.

Stack2vec adopts the unordered version of doc2vec (i.e., Distributed Bag of Words version of Paragraph Vector) since there is no fixed order for the subtrees in S. Similar to the Skip-Gram model of word2vec [34], it is a shallow but wide neural network (Fig. 5). The input layer takes one-hot vectors of the CSTrees. For T_i, the input vector is $(0,...,0,1,0,...,0)^T$, whose component at i is 1 and the others are 0. The hidden layer consists of linear neurons; the number of the neurons

is the dimension of the embedding space, which can be specified by the user. The output is a softmax layer which predicts the probability of each subtree's "frequency" in a CSTree. For example, the input T_i is trained with the aggregated weights of subtree `A - J` (Fig. 4 (c)) in the output layer.

Adaptive Moment Estimation (ADAM) [26] is employed in stack2vec to train the network with adaptive learning rates. Negative sampling [34] is used for fast training of stack2vec. After training, the output $\boldsymbol{x}_i$ from the hidden layer for each CSTree T_i is used as its embedding vector, which encodes the structural information of T_i.

5 Visual Training for Active Anomaly Detection

With the embedding vectors $\{\boldsymbol{x}_i\}$ of the CSTrees $\{T_i\}$, an unsupervised learning algorithm (e.g., Local Outlier Factor [7]) could be employed to find the outliers. However, the predictive performance of purely unsupervised anomaly detection methods is not always satisfactory [17]. Anomalous execution patterns cannot be determined only by gauging the density distribution in the embedding space. For example, an initialization subroutine inside a function `A` (e.g., `smalloc` of `compute` for allocating memory) can be rare since it is only invoked in the first execution of `A` in each HPC node, but it is an expected normal behavior by source code design. Injecting some domain knowledge into the process can help resolve these misconceptions.

To empower the model to accept additional labels from the user, one solution is to convert it to a semi-supervised binary classifier for normal and abnormal CSTrees [15]. However, our datasets are extremely unbalanced with just a few anomalies. Furthermore, it is difficult to provide a set of comprehensive abnormal patterns since most of them are unknown to the user. As a result, there will be insufficient negative labeling for the training of the abnormal class.

Our approach combines both the semi-supervised and the unsupervised learning strategies using a One-Class Support Vector Machine (OCSVM) [43]. Although being an unsupervised anomaly detection algorithm, OCSVM is adopted due to its capability of utilizing the labeling information to improve the model generation.

5.1 Anomaly Detection using One-Class Support Vector Machine

An OCSVM may be viewed as a regular two-class Support Vector Machine where all the training data are assumed to be positive (normal). OCSVM attempts to learn a decision boundary that separates the majority of the data from the origin. The data outside the decision boundary are considered outliers. For example, the dashed circle in Fig. 6 (a) approximates the decision boundary for outlier detection in the LAMMPS [42] [41] dataset (Section 6.1).

$$\begin{aligned} \min_{\boldsymbol{w},\boldsymbol{\xi},\rho} \quad & 0.5 \cdot \|\boldsymbol{w}\|^2 - \rho + \frac{1}{C}\sum_{i=1}^{n} u_i \xi_i \\ s.t. \quad & \boldsymbol{w}^T \Phi(\boldsymbol{x}_i) - \rho \geq -\xi_i, \;\; \xi_i \geq 0 \end{aligned} \tag{1}$$

OCSVM solves the optimization problem given in Eq. 1, where $\boldsymbol{w}$ and ρ are parameters of the decision function to be learned. ξ_i is the slack variable that allows a point to lie outside the decision boundary. Φ is the kernel function and C is a regularization parameter. The weighted OCSVM [4] introduces the weight u_i for $\boldsymbol{x}_i$ in Eq. 1. They are initialized as 1 in the model.

Although OCSVM is not capable to be trained by labeling, our approach utilizes the labels to improve the model though manipulating the weights $\boldsymbol{u}$. On the one hand, if $\boldsymbol{x}_i$ is labeled as negative (abnormal) by the user, the weight u_i is set to 0, which is equivalent to removing it from the training set. On the other hand, if $\boldsymbol{x}_i$ is confirmed by the user as positive (normal), we can emphasize it in the model by increasing u_i to a larger weight. For example, setting u_i to be larger than the estimated number of outliers (e.g., $0.02|\mathscr{T}|$) makes sure that $\boldsymbol{x}_i$ will not be regarded an anomaly. By updating it with the labeling, the final OCSVM will learn to model only the normal points. Other points unfamiliar to the OCSVM will be regarded as anomalies.

For $\boldsymbol{x}$, its decision function value $score(\boldsymbol{x}) = \boldsymbol{w}^T \Phi(\boldsymbol{x}) - \rho$ can be used as its anomaly score. $\boldsymbol{x}$ is abnormal if $score(\boldsymbol{x})$ is negative, otherwise it is normal. To support visual analysis task **T1**, the scores of all CSTrees are visualized as points in a scatter plot (Fig. 6 (a)). Red color indicates a negative score, which denotes points of special interest. Different projection methods such as Multidimensional Scaling (MDS) [30], or t-Distributed Stochastic Neighbor Embedding (t-SNE) [33] can be employed to calculate the point positions. Zooming in/out is allowed in the scatter plot to help the exploration of regions in which points are densely distributed.

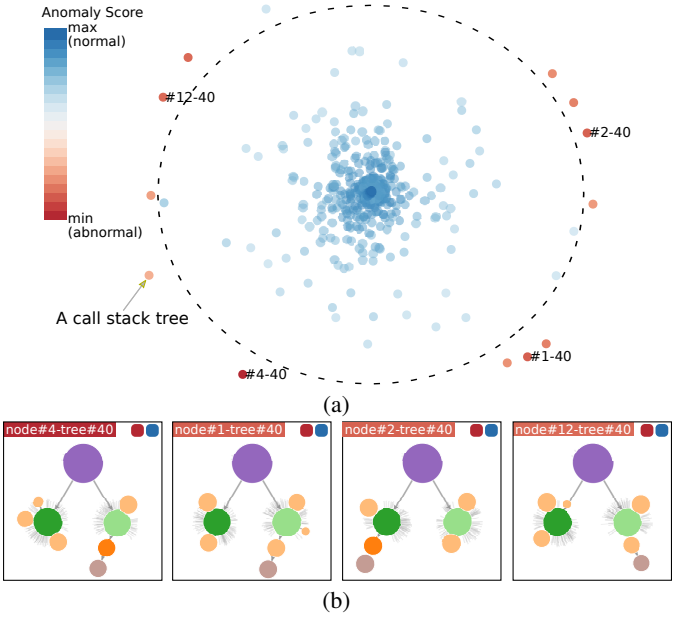


Fig. 6. (a) The MDS projection of the embedding vectors $\{\boldsymbol{x}_i\}$ of the CSTrees $\{T_i\}$. A One-Class SVM is employed to calculate the anomaly scores of $\{\boldsymbol{x}_i\}$. We manually draw the dashed circle to illustrate the approximate decision boundary of the One-Class SVM. (b) The candidate anomaly list shows the summary structures of CSTrees with lowest anomaly scores. The user can label each candidate as normal/abnormal using the blue/red button at the top right in each view.

5.2 Active Learning with Interactive Labeling

For CSTrees, labeling is labor intensive since the user needs to manually examine the structures in full detail. Instead of labeling all CSTrees or randomly labeling a subset of CSTrees, a more efficient labeling strategy is needed to find those samples that help optimize the model. Active learning [45] provides a solution to focus on the most informative subset of data. For a binary classifier, it asks the user to label the uncertain samples on the decision hyperplane between two classes. Since in our approach the OCSVM has only one positive class, the CSTrees with the most negative anomaly scores can be considered as the "support vectors". They will be presented to the user and visualized in the candidate anomaly list (Fig. 6 (b)). The summary structures of these CSTrees are visualized to provide insights into their patterns.

To support visual analysis task **T2**, candidates are ranked according to their anomaly scores. This allows the user to focus on the most anomalous CSTrees in a large dataset. The user can label an interesting CSTree in this list after detailed visualization (Section 5.3).

5.3 Visual Investigation and Verification

For an informed labeling of CSTrees, the user needs to be able to investigate their detailed execution behavior (visual analysis tasks **T3** and **T4**). For this purpose we have devised both a subtree (**T3**) and a timeline visualization (**T4**) to support the exploration of the structural and temporal pattern of a selected CSTree, respectively.

During the labeling, the user can select a CSTree of interest from the scatter plot (Fig. 6 (a)) or the candidate list (Fig. 6 (b)). Further, the user may also explore the average pattern of a group of CSTrees by selecting a region in the scatter plot using a lasso tool (hand drawn region in Fig. 9 (a)). The center CSTree of the selection is then visualized.

5.3.1 Top Subtree Visualization

For a selected CSTree T, the complete structure is presented (see the left two views of Fig. 7). The area and color of a tree vertex encode the vertex weight and function name, respectively. A force-directed algorithm [14] is used to make the tree layout compact.

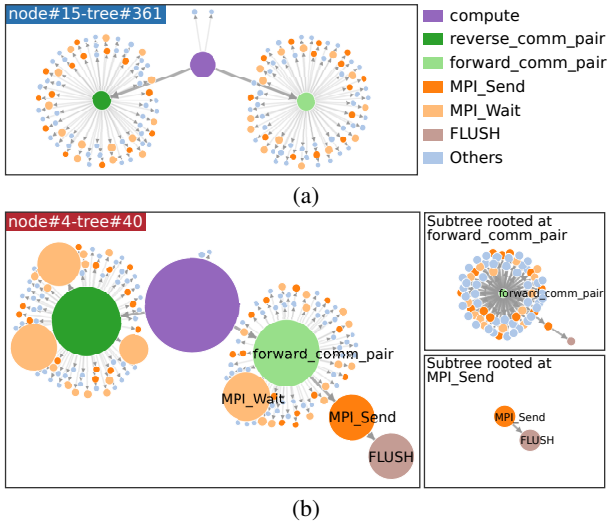


Fig. 7. The user can select CSTrees of interest to view their complete structures and important subtrees. (a) Structural pattern of a regular CSTree. There is no significant anomalous subtree. (b) A candidate CSTree with unusual patterns. The subtrees rooted at `forward_comm_pair` and `MPI_Send` are the top two substructures that make the CSTree abnormal.

From our case studies we found that users typically determine the reason for a candidate to be an anomaly by searching and identifying the unusual substructures manually. This exploration can be time consuming when the size of the CSTree is large. Extracting and visualizing the top anomalous subtrees can help shorten this time overhead. For this purpose we calculate the importance score of each subtree in T. This score describes the contribution of a subtree for making T anomalous.

We first compute the importance scores for all of the dimensions in the embedding vector $\boldsymbol{x}$ of T, and then translate the scores from the embedding space to the subtree space $\{s\}$. A straightforward and fast strategy to calculate the score vector $\boldsymbol{f}$ of $\boldsymbol{x}$ is to find the differences between $\boldsymbol{x}$ and its k-nearest neighbors $kNN(x)$. For each component $x^{(j)}$ of $\boldsymbol{x}$, its importance score $f^{(j)}$ is defined as the summed differences to the same component of its neighbors: $f^{(j)} = \sum_{y \in kNN(x)} |x^{(j)} - y^{(j)}|$. Therefore, from $\boldsymbol{f}$ we can learn which dimensions of $\boldsymbol{x}$ are the most different in the neighborhood.

To convert the importance score in the embedding space to the original subtree importance, we can take advantage of the relation that exists between the embedding $\boldsymbol{x}$ and the subtrees $\{s\}$ in stack2vec (Fig. 5). By inputting $\boldsymbol{f}$ to the hidden layer of stack2vec, the importance score vector for the subtrees is given by the output in the last layer.

With the importance scores of the subtrees calculated, the user can set a score threshold in the visual interface to focus on the top subtrees. These subtrees are shown in the Top Subtree Visualization (see the right views of Fig. 7 (b)) to provide insights into the anomalous substructures of the selected CSTree T. Because a substructure may occur in multiple parts of T with different vertex weights, we visualize a subtree with uniform vertex sizes to better focus on its structural pattern.

5.3.2 Timeline and Message Visualization

Similar to the visual encodings of Jumpshot [55] and Vampir [28], we show the function invocations and communications within a timeline visualization. The selected CSTree is visualized as a top down stack (e.g., the stack in `node#15` in Fig 8 (a)) along the time axis. Each invoked function in the CSTree is shown as a rectangle whose horizontal start and end positions encode its entry and exit timestamps in the stack, respectively. The vertical position of the function shows its position in the call stack. For example, in Fig 8 (a), the topmost `compute` is the root function of the selected stack. The two green functions in the second row are the children functions called by `compute`.

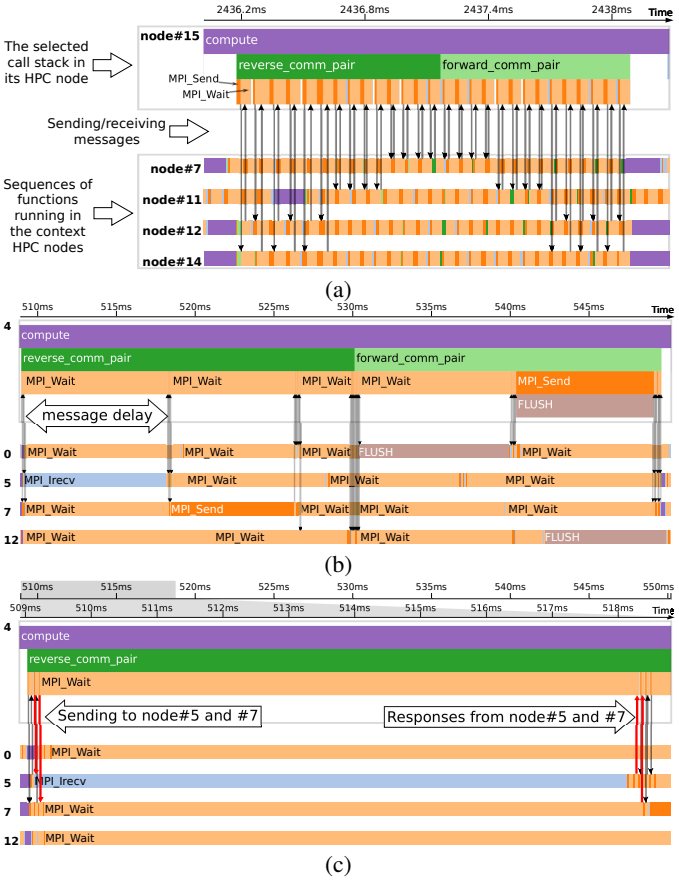


Fig. 8. The timeline view shows the call stack in a top-down manner along the time axis. Functions and messages are visualized as rectangles and arrowed lines, respectively. (a) The timeline visualization of the same CSTree in Fig. 7 (a). (b) The time series of the anomalous CSTree in Fig. 7(b). (c) The user can zoom into a time window to examine the events and messages in more details.

Because communication latency is a typical factor of performance degradation, messages are visualized as vertical arrowed lines from their sending nodes to the receiving nodes at their timestamps (see black lines between `node#15` and context HPC nodes in Fig. 8 (a)). To provide communication context, the nodes which communicate with the selected CSTree are visualized as sequences of function executions. For example, `node#7` in Fig. 8 (a) shows the sequence of functions running in it along the time axis.

Since the function events may occur very intensely in a small time interval, zooming into an interesting time window can help investigate any detailed activities of the nodes. We provide level-of-detail exploration in the timeline, which allows the user to view the functions and messages in different temporal granularity. For example, Fig. 8 (c) shows the zoomed timeline of Fig. 8 (b) between $509ms$ to $519ms$.

6 Case Studies

We conducted case studies with three scientific collaborators (SC1, SC2 and SC3), all of whom were physicists. SC1 and SC2 were from a team working on the LAMMPS [42] [41] application, which is a parallel molecular dynamics simulator in the HPC cluster. SC3 was interested in the NWCHEM [50] application, which is another scientific program for molecular simulation. The participants collected the trace events of the HPC cluster via TAU instrumentation [46]. They wanted to use our system to find anomalous runtime behaviors in the function

executions. None of the participants were experts in visualization or machine learning. The summaries of their datasets are shown in Table 2.

Dataset	Case 1 & 2	Case 3
Total number of CSTrees	5,792	1,280
Total number of vertices	879,332	1,078,804
Total number of messages	323,780	646,683
Number of unique subtrees (Subtree corpus dimension)	901	843
Maximum CSTree depth	5	4
Embedding dimensions	128	128

Table 2. Summary of the forests constructed in the cases.

Before the study, we had a number of thorough discussions with the domain scientists to learn about functions of interest and possible anomalous behavior patterns. Each case study started with a training session to introduce our system. We then asked each participant to use the system for their datasets. Finally, an interview session was conducted to gather evaluations and subjective feedback.

6.1 Case Study 1: Investigating Functions with Communication Delays

SC1 ran LAMMPS on 16 HPC computation nodes which were supported by 16 I/O nodes. He would like to learn about the factors and types of the execution delay. Especially, SC1 was interested in `compute`, a function for the major computational task with frequent message communications. `compute` was called 5,792 times in the HPC cluster (Table 2). We constructed a forest with each tree representing an invoked execution of `compute`. The WL labeling extracted 901 unique subtree structures from the forest. We pre-computed the vector representation $\{\boldsymbol{x_i}\}$ of the CSTrees $\{T_i\}$ using stack2vec with an embedding size of 128. All CSTrees were unlabeled at the start of the analysis.

SC1 first trained OCSVM to detect the anomalous CSTrees. The learning results showed that most anomalous candidates were located on the border of the scatter plot distribution (Fig. 6 (a)).

SC1 chose CSTree `#4-40` from Fig. 6 (b), which was the first candidate in the list. Here, the code "`#4-40`" means that it was the `#40` invocation of `compute` in `node#4`. To compare it with a regular call stack pattern, he also selected a normal CSTree `#15-361` in the center of the scatter plot. Fig. 7 (a) shows that the sizes of all vertices in `#15-361` are small, suggesting short function executions. In contrast, SC1 observed that the vertices of `MPI_Wait`, `MPI_Send`, and `FLUSH` in `#4-40` were unusually large (see the left view of Fig. 7 (b)). As expected, the corresponding subtrees were identified as anomalous in the Top Subtree Visualization (see the right two windows in Fig. 7 (b)).

To learn about the temporal pattern and message communication, SC1 examined the Timeline Visualizations. He noticed from the time axes that execution of `#15-361` (Fig. 8 (a)) took only about 2*ms* while `#4-40` (Fig. 8 (b)) took 20 times longer (about 40*ms*). `#15-361` showed a normal communication pattern, where `MPI_Send` and `MPI_Wait` were invoked alternately multiple times (see dark and light orange functions in `node#15` in Fig. 8 (a)). They sent/received messages (see arrowed lines in Fig. 8 (a)) to/from the context nodes with regular and short execution times. While in the timeline of `#4-40`, SC2 found that there were communication delays of the functions `MPI_Wait` and `MPI_Send` (see `node#4` of Fig. 8 (b)), which made the total execution time of `#4-40` much longer than that of `#15-361`.

SC1 zoomed into the time window of the first long `MPI_Wait` function in `#4-40`, as shown in Fig. 8 (c). From the messages (see highlighted red lines in Fig. 8 (c)), he learned that `#4-40` sent two messages to `node#5` and `node#7` around 509*ms* and then waited for their response in `MPI_Wait`. However, at that time, `node#5` and `node#7` were also delayed in their `MPI_Irecv` and `MPI_Wait`, respectively. This formed a chain of waiting, which took 10*ms* until `node#5` and `node#7` sent messages back to `#4-40`. SC1 pointed out that this indicated a problem with the scheduling strategy. By exploring the remaining portion of `#4-40`, SC1 also noticed a long `FLUSH` (see `FLUSH` in `node#4` in Fig. 8 (b)), causing the waiting of `node#5` and `node#7` consequently.

Finally, after examination of other candidate CSTrees (Fig. 6 (b)), SC1 found that the similar pattern of delay chain also occurred in other nodes around the same time period, such as in `#1-40`, `#2-40`, and `#12-40`. He concluded that he needed to adjust the machine configuration to avoid these types of delay situations in the future.

SC1 went on and labeled all those CSTrees as abnormal and exported them for his scheduling strategy analysis. He trained the model again with his labeling and saved the model for future prediction of anomalous executions in LAMMPS applications.

6.2 Case Study 2: Finding Potential Cluster Anomalies

SC2, who was from the same team of SC1, also focused on the `compute` function in the LAMMPS dataset. He used the OCSVM model from the last study, since it was improved by the labeling of SC1. SC2 focused on collective anomalies. He mentioned that a group of anomalies with similar patterns were likely to be labeled as normal by the automatic algorithm. This was because the OCSVM could only detect an outlier when it had a different pattern with respect to its neighbors.

At the onset of the exploration, SC2 found that most normal points were projected in the scatter plot center, yielding a display too crowded to observe any distribution patterns. He therefore switched to t-SNE projection (Fig. 9 (a)), which reduced the tendency to crowd points together [33]. He explored the points in a cluster far away from others (see the highlighted points in Fig. 9 (a)) and noticed their distinct structural patterns. He selected the cluster using the lasso tool; the center of the selection was shown in the structure and timeline visualizations.

Different from the structural pattern of a common CSTree (e.g., `#15-361` in Fig. 7 (a)), the CSTree of the cluster center (see tree `#0-161` in Fig. 9 (b)) invoked an abundance of tiny `ev_tally` functions. The Top Subtree Visualization (see the right two views in Fig. 9 (b)) also indicated that the substructures which contained `ev_tally` made this CSTree different from others in the forest. After investigating the timeline view (Fig. 9 (c)), SC2 told us that he was confused why there were frequent invocations of `ev_tally`s after the exit of `forward_comm_pair` in `compute`.

By zooming into the timeline (Fig. 9 (d)), SC2 noticed that `ev_tally` was also invoked intensively in the context nodes at the same time, as shown in the pink strips in node `#1`, `#3`, `#4`, and `#8` in Fig. 9 (d). He mentioned that he was expecting lots of communications between `ev_tally`s of different HPC nodes. However, he did not observe any message from `ev_tally`s in the visualization. He suggested that `ev_tally`s might use other communication means (e.g., shared memory), which was not recorded in the form of trace events.

After exploring other points in the selected cluster, SC2 found that they had similar patterns with lots of `ev_tally`s. He labeled all CSTrees in this cluster as anomalies (Fig. 9 (e)), since he wanted to export them for further analysis with machine logs and source code. He trained the model again; the updated results showed that all the CSTrees in the selected cluster changed from normal to abnormal.

6.3 Case Study 3: Identifying Functions with Unusual Execution Sequences

SC3 was one of the developers of the NWCHEM application on the HPC cluster. He focused on the `MD_NEWTON` function for solving Newton's equations of motion [50]. The application executed in 5 HPC nodes and `MD_NEWTON` was called 1,280 times. A forest was constructed in which each CSTree is an execution of `MD_NEWTON`. We represented each CSTree by a embedding vector of length 128. All CSTrees at the beginning of the study were unlabeled.

After training OCSVM with the initial $\boldsymbol{u}$ (Eq. 1), the candidates with low anomaly scores were shown as red points in Fig. 1 (a). SC3 found that the first executions of `MD_NEWTON` in each HPC node were listed as the top anomaly candidates (see `#0-0`, `#1-0`, `#2-0`, and `#3-0` in Fig. 1 (b)). He selected `#0-0` for a detailed investigation. From the Top Subtree View (Fig. 1 (c)), SC3 learned that the substructure rooted at the `MD_SHAKE` function was one major reason for `#0-0` being anomalous. In the timeline (Fig. 1 (d)), he zoomed into `MD_SHAKE` and found that it called lots of different `MPI` functions, such as `MPI_Type_size`.

After exploring other normal CSTrees in the center of the scatter plot, SC3 found that usually `MD_SHAKE` had few `MPI` children functions. He explained that the first execution (`#X-0`) in each HPC node would

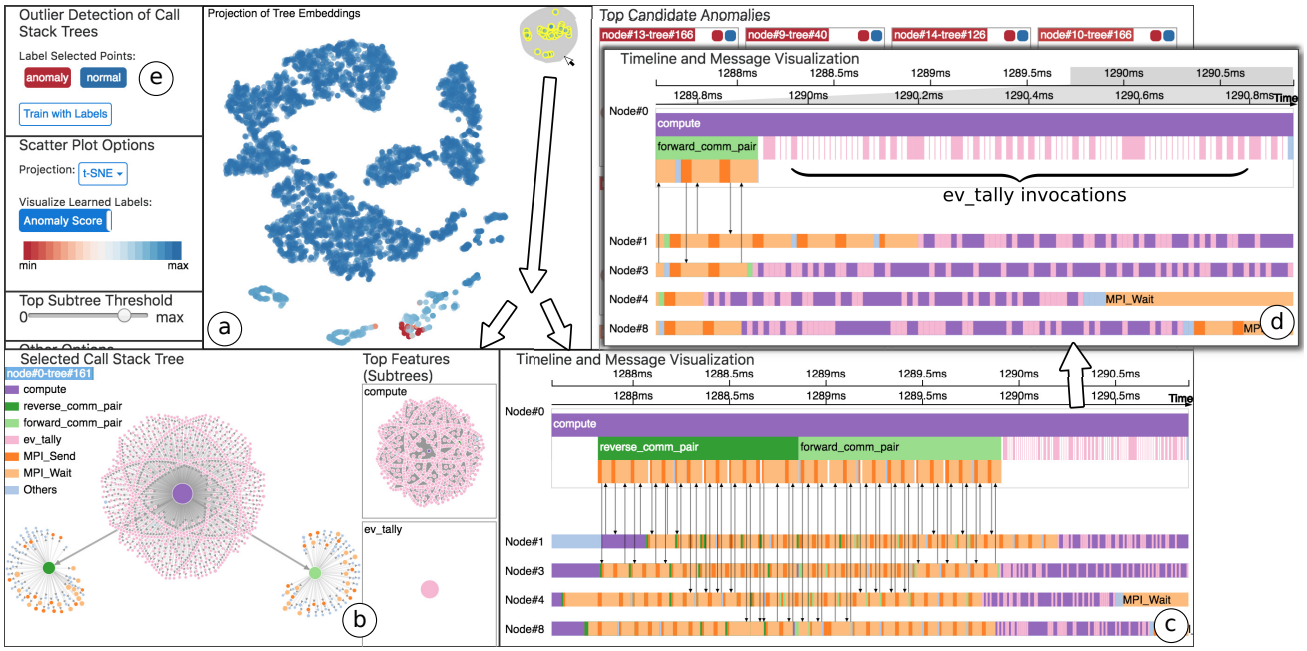


Fig. 9. Anomalous execution detection of `compute` in LAMMPS dataset. (a) T-SNE projection of the embedding vectors of the CSTrees. SC2 selected a normal cluster far away from other points. (b) The structure of the cluster center (`#0-161`) was visualized. The subtrees with `ev_tally` were unexpected structural patterns. (c) The timeline showed that a lot of `ev_tally`s were invoked at the end of `compute`. (d) SC2 zoomed into the timeline and noticed that there was no message from the `ev_tally`s. (e) Finally, SC2 labeled all the CSTrees in the selected cluster as abnormal.

invoke a sequence of `MPI` functions for the initialization of the communication between nodes. He expressed that although this behavior was rare, those functions were executed as expected. He labeled `#0-0`, `#1-0`, `#2-0`, and `#3-0` as normal (Fig. 1 (e)), which increased their weights $\boldsymbol{u}$ as positive samples in OCSVM model. By training again, SC3 found that the anomaly scores of those CSTrees became positive.

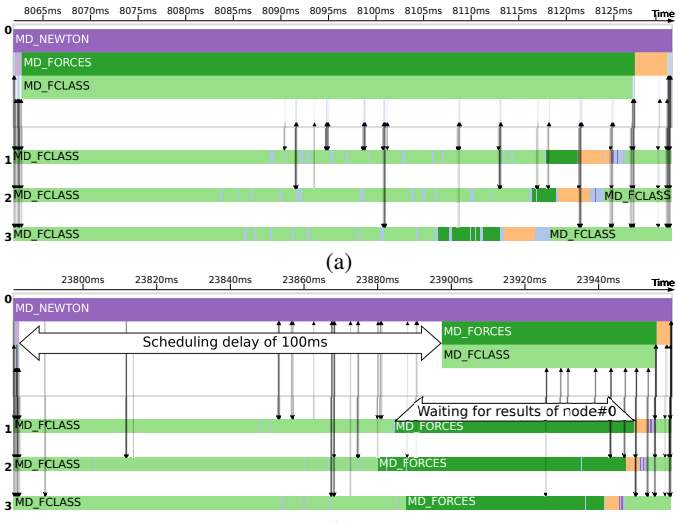


Fig. 10. (a) A normal timeline of CSTree `#33` in `node#0`. (b) The timeline of CSTree `#192` in `node#0` suggested the late invocation of `MD_FORCES`. It also caused the latencies of the `MD_FORCES` functions in other nodes.

SC3 continued to explore the remaining candidates. He selected `#0-192` in Fig. 1 (b), which was the `#192` execution of `MD_NEWTON` in `node#0`. The time axis of `#0-192` (Fig. 10 (b)) indicated that it took about 170*ms*, which was 100*ms* longer than a normal execution (e.g., `#0-33` in Fig. 10 (a)). He mentioned that `MD_NEWTON` was expected to invoke `MD_FORCES` immediately (Fig. 10 (a)), while `#0-192` called `MD_FORCES` after waiting about 100*ms* (Fig. 10 (b)).

After careful investigation, SC3 pointed out that `#0-192` also caused a delay of its context executions in `node#1`, `#2`, and `#3` in Fig. 10 (b). He pointed that the context HPC nodes were waiting for the simulation result of `#0-192`, as indicated in the delayed `MD_FORCES` functions of `node#1`, `#2`, and `#3` in Fig. 10 (b). He concluded that this indicated a scheduling problem of the application. SC3 labeled them as anomalies and trained OCSVM again. He examined and noticed that other CSTrees with similar patterns turned abnormal after his update.

6.4 Quantitative Analysis

Using the datasets and scenarios described above, we evaluate the performance and complexity of stack2vec and active anomaly detection.

6.4.1 Performance of Stack2vec and Active Anomaly Detection

We begin by comparing the performance of the stack2vec embedding with the conventional feature representation of function execution. From our discussions with the scientists we learned that they usually construct a feature vector for each execution using the runtime information, including execution time, exit time, message number, and node id. We denote this method as "Time" in Table 3. Then, to remove the influences of the outlier detection algorithms in our comparison, we employed two different anomaly detection methods: Local Outlier Factor (LOF) [7] and the original One-Class Support Vector Machine (OCSVM) [43]. To establish a baseline ground truth we asked the three scientists SC1-3 to label all data after the case studies. We tested the areas under both the Receiver Operating Characteristic curve (ROC) and Precision-Recall curve (PR) of different anomaly detection pipelines, including Time+LOF, Stack2vec+LOF, Time+OCSVM, and Stack2vec+OCSVM. As shown in the top four rows in Table 3, stack2vec has better performance than Time. This is because stack2vec is able to encode structural information of the complete call stack.

We also compare the unsupervised anomaly detection approaches with our OCSVM with active labeling ("AL" in Table 3). Stack2vec+AL improves the performance significantly in both datasets. This is because the automatic outlier detection approach only allows the user to set a hyper-parameter such as the percentage of outliers for the overall dataset. This tends to be sub-optimal since the domain knowledge

Methods	Case 1 & 2		Case 3	
	ROC	PR	ROC	PR
Time+LOF	0.970	0.792	0.919	0.575
Stack2vec+LOF	0.976	0.854	0.959	0.643
Time+OCSVM	0.958	0.760	0.919	0.557
Stack2vec+OCSVM	0.968	0.808	0.955	0.612
Stack2vec+SVM	0.994	0.921	0.990	0.716
Stack2vec+AL	0.993	0.909	0.990	0.732

Table 3. Comparison of areas under curves of both ROC and PR for different representation approaches and anomaly detection strategies.

is especially critical for identifying the specific runtime behavior in performance analysis, such as the clustered anomalies in Case 2 and the unusual but normal initialization subroutines in Case 3. For comparison with a supervised model, we also test the SVM, which has a close performance to AL. However, the labeled dataset for SVM training is usually not available in most anomaly detection application.

6.4.2 Complexity of Stack2vec

Since the complexity of our anomaly detection is the same with the original OCSVM, we only discuss stack2vec in this section.

In our accelerated WL algorithm, a vertex v appears only once in its parent's signature in the label propagation stage, and is visited once more in its label compression stage. As a result, the total complexity for h iterations is linear to the number of vertices in the forest $O(h|\mathscr{T}||V|)$. It is much faster than the original WL algorithm, which has a complexity of $O(h|\mathscr{T}|^2|V|)$ for a forest. In our algorithm, h is set according to the maximum depth of the CSTrees, as mentioned in Section 4.1. Since the call stack depth in parallel computing is limited (e.g., < 20), our algorithm in practice is much faster.

During our study, we found that the sizes of subtree corpora generated by WL algorithm (see unique subtree number in Table 2) were very small compared with the word corpus in a Natural Language Processing (NLP) application. SC1 mentioned that this might be because the CSTrees were generated from the same source code, limiting the diversity of subtree patterns. Due to this reason, the scalability of stack2vec is better than other doc2vec application in NLP. For a small subtree corpus, we can also directly use the bag-of-subtree vector as the input for anomaly detection without neural embedding.

To save time for interactive exploration and labeling, the WL algorithm and neural tree embedding in stack2vec are pre-computed.

6.4.3 Parameter Settings

Since the subtree corpus is usually not large, stack2vec uses an embedding size of 128, which we found is sufficient for the representation of a CSTree structure. For anomaly detection, we use a radial basis function kernel (Gaussian) in OCSVM. The regularization parameter C in Eq. 1 is regarded as the estimated number of outliers [43]. Based on the anomaly frequencies we observed in our datasets, we set $C = 0.02|\mathscr{T}|$.

6.5 User Feedback

We evaluated the learning cost and usability of our visual interface both in the training session and in the interview session, respectively.

6.5.1 Learning Cost

In the training session, our instructor gave a 15-minute demo to explain our algorithm and visualizations. Then SC1, SC2 and SC3 practiced our system with the help of the instructor. They were free to stop practicing whenever they felt ready. To evaluate the learning progress, each participant was given a test which consisted of 6 exercises, which tested user understanding of the proposed visualizations. For example, "Find and select the top anomalous CSTree in the candidate list". We observed that all participants practiced for less than 10 minutes. In the tests, the participants were able to respond to all exercises quickly as expected without any help from our instructor. The participants gave an average rating of 4.67 for the learning cost of our system (1 = very hard, 5 = very easy). All of the participants mentioned that the views were easy to follow since the tree and timeline visualizations are commonly-used by the existing performance analysis tools [22] [13].

6.5.2 Usability

In the interview session, all participants rated the usefulness of our system at 5 (1 = very useless, 5 = very useful). We then asked them to give detailed evaluations of how our system supported their visual analysis tasks. SC1 mentioned that the tree structure visualization provided him intuitive understanding of the function call paths (**T3**). He also commented that, "Usually the machine learning process is a black box; it is hard to understand what is learned. But the Top Subtree Visualization allowed me to get insights into what sub-structures make a call stack anomalous". SC2 suggested that our scatter plot allowed him to explore and identify clustered anomalies, which were usually ignored by an automatic outlier detection algorithm (**T1**). SC3 noted that our system answered why the performance fluctuated in his NWCHEM application. He commented that ranking candidates by anomaly scores enabled him to only focus on the top anomalies among the large number of function executions (**T2**). He also told us that the timelines allowed him to quickly determine where the latency came from by examining the execution duration and message distribution (**T4**).

The participants were also asked to compare our system with their usual tools. All of them indicated that usually they could only select candidate anomalies by their long execution times. Furthermore, they had to manually locate the time window of a candidate in Jumpshot [55] to visualize the detailed call paths and messages. In contrast, they praised our framework as more efficient in both identifying and understanding the anomalous behaviors since it is based on the CSTree representation. SC3 commented that our learning strategy allowed him to flexibly adapt the model for special behaviors in HPC. He also mentioned that the active labeling saved him a lot of effort, since he only needed to inspect the potential anomalous executions.

6.6 Discussion and Generalizability

Our approach can be modified based on the actual cases for better performance. First, although our CSTree representation ignores the order of the function invocations, it still allows to model the temporal order by adding directed edges between the sibling functions. Second, cluster-based anomalies could be detected by employing a clustering algorithm to supplement the OCSVM. In this case, clusters whose average patterns are unusual would be reported as candidate anomalies.

We would like to emphasize that our framework is not restricted to call stacks only. Rather, it applies to any application where the data can be represented as a directed tree structure. For example, we might use it for the identification of interesting branched lineages in social or citation networks. Here, each person (paper) is a vertex and each social link (citation) is modeled as a directed edge between the persons (papers) starting at some root person (paper). The weights might be some importance metric defined on each person (paper). Our framework could then identify anomalous (sub)structures that point to interesting personalities or papers and key connections for either.

Our approach is also able to be extended to cyclic graph structures. In this case, the substructure features can be extracted by the original WL labeling algorithm, which deals with the general graph.

7 Conclusion

We described a visual analytics approach for detecting anomalous executions in HPC clusters. For this purpose we created a CSTree representation and devised a stack2vec embedding method to model the runtime behavior. Moreover, we proposed an active labeling strategy that integrates anomaly detection and user input of domain knowledge during the visual analysis.

For future work, our approach can be integrated with the machine log analysis and source code study for a complete analysis pipeline of the execution scheduling and code design optimization.

Acknowledgments

We thank Shinjae Yoo, Wen Zhong and Jianing Yan for helpful discussions. This research was partially supported by NSF grant IIS 1527200, BNL LDRD grant 16-041, ECP CODAR project 17-SC-20-SC, and the MSIP (Ministry of Science, ICT and Future Planning), Korea, under "IT Consilience Creative Program (ITCCP)" supervised by NIPA.

REFERENCES

[1] N. Abe, B. Zadrozny, and J. Langford. Outlier detection by active learning. In *Proceedings of the 12th ACM SIGKDD*, pp. 504–509. ACM, 2006.

[2] L. Adhianto et al. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[3] D. C. Arnold, D. H. Ahn, B. R. De Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Stack trace analysis for large scale debugging. In *IPDPS 2007. IEEE International*, pp. 1–10. IEEE, 2007.

[4] M. Bicego and M. A. Figueiredo. Soft clustering using weighted one-class support vector machines. *Pattern Recognition*, 42(1):27–32, 2009.

[5] G. Blanchard, G. Lee, and C. Scott. Semi-supervised novelty detection. *Journal of Machine Learning Research*, 11(Nov):2973–3009, 2010.

[6] K. M. Borgwardt and H.-P. Kriegel. Shortest-path kernels on graphs. In *IEEE ICDM*, pp. 8–pp. IEEE, 2005.

[7] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. Lof: identifying density-based local outliers. In *ACM SIGMOD*, vol. 29, pp. 93–104, 2000.

[8] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15, 2009.

[9] W. Chen, J. Xia, X. Wang, Y. Wang, J. Chen, and T. Gu. Relationlines: Visual reasoning of egocentric relations from heterogeneous urban data. *ACM Transactions on Intelligent Systems Technology*, 2018.

[10] S. Cheng and K. Mueller. Improving the fidelity of contextual data layouts using a generalized barycentric coordinates framework. In *Visualization Symposium (PacificVis), 2015 IEEE Pacific*, pp. 295–302. IEEE, 2015.

[11] S. Cheng and K. Mueller. The data context map: Fusing data and attributes into a unified display. *IEEE TVCG*, 22(1):121–130, 2016.

[12] S. Cheng, W. Xu, and K. Mueller. Colormapnd: A data-driven approach and tool for mapping multivariate data to color. *IEEE TVCG*, pp. 1–1, 2018. doi: 10.1109/TVCG.2018.2808489

[13] N. Ezzati-Jivan and M. R. Dagenais. Multi-scale navigation of large trace data: A survey. *Concurrency and Computation: Practice and Experience*, 29(10), 2017.

[14] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Softw. Pract. Exper.*, 21(11):1129–1164, Nov. 1991. doi: 10.1002/spe.4380211102

[15] J. Gao, H. Cheng, and P.-N. Tan. Semi-supervised outlier detection. In *Proceedings of the 2006 ACM symposium on Applied computing*, pp. 635–636. ACM, 2006.

[16] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.

[17] N. Görnitz, M. M. Kloft, K. Rieck, and U. Brefeld. Toward supervised anomaly detection. *Journal of Artificial Intelligence Research*, 2013.

[18] A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. In *Proc. ACM SIGKDD*, pp. 855–864, 2016.

[19] Q. Guan, D. Smith, and S. Fu. Anomaly detection in large-scale coalition clusters for dependability assurance. In *Proc. IEEE High Performance Computing (HiPC)*, pp. 1–10, 2010.

[20] Z. He, X. Xu, and S. Deng. Discovering cluster-based local outliers. *Pattern Recognition Letters*, 24(9-10):1641–1650, 2003.

[21] F. Heimerl, S. Koch, H. Bosch, and T. Ertl. Visual classifier training for text document retrieval. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2839–2848, 2012.

[22] K. Isaacs, A. Giménez, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, and P.-T. Bremer. State of the art of performance visualization. *Proc. EuroVis*, 2014.

[23] H. Janetzko, F. Stoffel, S. Mittelstädt, and D. A. Keim. Anomaly detection for visual analytics of power consumption data. *Computers & Graphics*, 38:27–37, 2014.

[24] B. Karran, J. Trumper, and J. Dollner. Synctrace: Visual thread-interplay analysis. In *IEEE Working Conf. on Software Visualization (VISSOFT)*, pp. 1–10, 2013.

[25] H. Kashima, K. Tsuda, and A. Inokuchi. Marginalized kernels between labeled graphs. In *Intern. Conf. on Machine Learning*, pp. 321–328, 2003.

[26] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[27] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[28] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The vampir performance analysis tool-set. In *Tools for High Performance Computing*, pp. 139–155. Springer, 2008.

[29] A. Knüpfer et al. Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In *Tools for High Performance Computing 2011*, pp. 79–91. Springer, 2012.

[30] J. B. Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27, 1964.

[31] Q. Le and T. Mikolov. Distributed representations of sentences and documents. In *Intern. Conf. on Machine Learning*, pp. 1188–1196, 2014.

[32] S. Lin, F. Taïani, T. C. Ormerod, and L. J. Ball. Towards anomaly comprehension: using structural compression to navigate profiling call-trees. In *Proc. ACM Symp. on Software visualization*, pp. 103–112, 2010.

[33] L. v. d. Maaten and G. Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[34] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Proc. NIPS*, pp. 3111–3119, 2013.

[35] C. Muelder, B. Zhu, W. Chen, H. Zhang, and K.-L. Ma. Visual analysis of cloud computing performance using behavioral lines. *IEEE Trans. on Visualization and Computer Graphics*, 22(6):1694–1704, 2016.

[36] A. Narayanan, M. Chandramohan, L. Chen, Y. Liu, and S. Saminathan. subgraph2vec: Learning distributed representations of rooted sub-graphs from large graphs. *arXiv preprint arXiv:1606.08928*, 2016.

[37] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal. graph2vec: Learning distributed representations of graphs. *arXiv preprint arXiv:1707.05005*, 2017.

[38] H. T. Nguyen, L. Wei, A. Bhatele, T. Gamblin, D. Boehme, M. Schulz, K.-L. Ma, and P.-T. Bremer. Vipact: a visualization interface for analyzing calling context trees. In *IEEE Workshop on Visual Performance Analysis*, pp. 25–28, 2016.

[39] M. Niepert, M. Ahmed, and K. Kutzkov. Learning convolutional neural networks for graphs. In *Intern. Conf. on Machine Learning*, pp. 2014–2023, 2016.

[40] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: Online learning of social representations. In *Proc ACM SIGKDD*, pp. 701–710, 2014.

[41] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of computational physics*, 117(1):1–19, 1995.

[42] S. Plimpton. Large-scale atomic/molecular massively parallel simulator. `https://github.com/CFDEMproject/LAMMPS`, 2018.

[43] B. Schölkopf, R. Williamson, A. Smola, J. Shawe-Taylor, and J. Platt. Support vector method for novelty detection. In *NIPS*, pp. 582–588, 2000.

[44] U. Schöning. Graph isomorphism is in the low hierarchy. *Journal of Computer and System Sciences*, 37(3):312–323, 1988.

[45] B. Settles. Active learning literature survey. Technical report, 2010.

[46] S. S. Shende and A. D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006.

[47] N. Shervashidze, P. Schweitzer, E. J. v. Leeuwen, K. Mehlhorn, and K. M. Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(Sep):2539–2561, 2011.

[48] C. Sigovan, C. Muelder, K.-L. Ma, J. Cope, K. Iskra, and R. Ross. A visual network analysis method for large-scale parallel i/o systems. In *IEEE IPDPS*, pp. 308–319, 2013.

[49] O. Tuncer et al. Diagnosing performance variations in hpc applications using machine learning. In *Proc. Supercomputing*, pp. 355–373, 2017.

[50] M. Valiev et al. Nwchem: a comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477–1489, 2010.

[51] J. Xia, W. Chen, Y. Hou, W. Hu, X. Huang, and D. S. Ebertk. Dimscanner: A relation-based visual exploration approach towards data dimension inspection. In *IEEE VAST*, pp. 81–90, 2016.

[52] C. Xie, W. Xu, S. Ha, K. Huck, S. Shende, H. Van Dam, K. K. Van Dam, and K. Mueller. Performance visualization for tau instrumented scientific workflows. 2018.

[53] P. Xu, H. Mei, L. Ren, and W. Chen. Vidx: Visual diagnostics of assembly line performance in smart factories. *IEEE Trans. on Visualization and Computer Graphics*, 23(1):291–300, 2017.

[54] P. Yanardag and S. Vishwanathan. Deep graph kernels. In *Proc. ACM SIGKDD*, pp. 1365–1374, 2015.

[55] O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward scalable performance visualization with jumpshot. *Int. J. High Perform. Comput. Appl.*, 13(3):277–288, Aug. 1999.