

Identifying Performance Inefficiencies of Parallel Program With Spatial and Temporal Trace Analysis

Zhibo Xuan^{ID}, Xin Sun^{ID}, Xin You^{ID}, Hailong Yang^{ID}, *Member, IEEE*, Zhongzhi Luan^{ID}, *Senior Member, IEEE*, Yi Liu^{ID}, and Depei Qian^{ID}

Abstract—Performance inefficiencies can lead to performance anomalies in parallel programs. Existing performance analysis tools either have a limited detection scope or require significant domain knowledge to use, which constrains their practical adoption to identify performance inefficiencies. In this paper, we propose *STAD*, a performance analysis tool for parallel programs that considers both spatial and temporal patterns within trace data. *STAD* captures the spatial communication patterns between processes using a spatial communication pattern graph. It then adopts a dynamic graph neural network-based unsupervised model to learn the evolving temporal patterns along the timeline. Additionally, *STAD* diagnoses the root causes of performance anomalies by exploiting the aggregated feature of anomalies along the call tree. Our evaluation results demonstrate that *STAD* can effectively detect performance anomalies with acceptable overhead and diagnose the root causes attributed to both the program itself and the running environment.

Index Terms—Anomalies, parallel programs, performance analysis, root cause.

I. INTRODUCTION

HARDWARE and software inefficiencies are prevalent pitfalls that can lead to severely unexpected performance degradations in domains such as scientific computing [1], [2], [3], [4], [5], [6], and intelligent computing [7], [8], [9]. Such performance inefficiencies often necessitate comprehensive performance analysis [10], [11], [12], [13], [14], [15], [16], [17] to pinpoint the root causes for optimizing target application effectively. Unfortunately, the culprits behind these performance inefficiencies are often hidden deeply within complex software stacks and varied program behavior patterns. Specifically, according to our investigations of previous literatures [5], [6], [7],

[10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], the performance inefficiencies can be roughly classified into two categories: *endogenous inefficiencies* and *exogenous inefficiencies*. *Endogenous inefficiencies* are caused by the program itself, such as poor scalability [12] and load imbalance [8], [22], whereas *exogenous inefficiencies* are caused by the running environment, such as hardware variance [2], [3], [4], [7], [23], [24] and system noise [5], [20], [25], [26]. Therefore, various performance analysis tools [2], [3], [4], [7], [8], [10], [11], [12], [13], [14], [15], [16], [17], [22], [23] have been proposed to detect the above inefficiencies.

Although the existing performance analysis tools can detect and diagnose specific performance inefficiencies, they still suffer from several limitations: **1) narrowed detection scope of performance inefficiencies** - Existing performance analysis tools are developed for detecting specific performance inefficiencies. Since users often have no foresight of the type of performance inefficiencies to be encountered, they are forced to try multiple tools to diagnose a particular performance inefficiency, which is time-consuming and inefficient; **2) expert knowledge required for pinpoint performance inefficiencies** - Existing performance analysis tools commonly embed in-depth domain knowledge to pinpoint specific performance inefficiencies, making it difficult to use for non-experts; **3) heavily relying on extensive labeled datasets to identify performance inefficiencies** - Especially machine learning-based performance analysis tools require labeled and extensive datasets for training models capable of identifying specific performance inefficiencies, which necessitates non-trivial engineering efforts for training dataset preparation. In addition, due to the richness and diversity of performance inefficiencies, the detection accuracy is far from satisfactory for model-based performance analysis tools.

Fortunately, we have observed the following phenomenon for performance inefficiency that can be utilized to address the above limitations. Specifically, ***inefficient program execution within a certain period should result in evolving abnormal performance characteristics across different processes for parallel programs***. For example, the hardware performance variance, which is a well-known exogenous inefficiency, can lead to evolving abnormal process communication and computation patterns that are different from processes running on normal nodes (details refer to Section IV-B). On the other hand, the load imbalance, which is a well-known endogenous inefficiency, can lead to evolving abnormal program behaviors across processes

Received 18 November 2024; revised 13 March 2025; accepted 25 April 2025. Date of publication 2 May 2025; date of current version 20 May 2025. This work was supported by National Key Research and Development Program of China under Grant 2023YFB3001801, in part by the National Natural Science Foundation of China under Grant 62322201, Grant U23B2020 and Grant U22A2028, in part by the Fundamental Research Funds for the Central Universities under Grant YWF-23-L-1121, Grant JKF-20240198 and Grant JK2024-58, and in part by the State Key Laboratory of Software Development Environment under Grant SKLSDE-2023ZX-05. Recommended for acceptance by D. Li. (Zhibo Xuan and Xin Sun contributed equally to this work.) (Corresponding author: Hailong Yang.)

The authors are with the State Key Laboratory of Software Development Environment, School of Computer Science and Engineering, Beihang University, Beijing 100191, China (e-mail: xzb@buaa.edu.cn; sxin2206@buaa.edu.cn; youxin2015@buaa.edu.cn; hailong.yang@buaa.edu.cn; 07680@buaa.edu.cn; yi.liu@buaa.edu.cn; depei.qian@buaa.edu.cn).

Digital Object Identifier 10.1109/TPDS.2025.3566735

with similar calling contexts, such as unexpected heavy communication and computation for few processes over a certain period (details refer to Section IV-C).

Based on the above observation, we advocate that performance inefficiency detection can be transformed into performance anomaly detection with root cause diagnosis. With such transformation, we can innovate the approach for detecting performance inefficiencies inspired by existing anomaly detection [1], [6], [27] techniques. For example, Variational Auto-Encoders (VAEs) have been proven effective for performance anomaly detection due to their capability of capturing the underlying distribution of the data with unsupervised learning. Specifically, VAE evaluates the reconstruction probability of a specific sample to identify performance anomalies. However, for reasonable distribution modeling, VAE requires a well-formed representation of target program behaviors, including communication and computation patterns. Fortunately, existing works [10], [12], [28] have adopted the graph-based representation to depict program behaviors with rich program semantics and patterns.

To better capture the performance inefficiencies for parallel programs, we propose a Spatial Communication Pattern Graph (SCPG) to capture the spatial patterns between processes in parallel programs, including communication pairs and execution segments with performance events. To further represent the evolving behaviors of the target program represented in SCPG, we adopt the dynamic graph neural network (DGNN) [1], [27], [29], [30] to capture the dependencies in both spatial and temporal dimensions. Specifically, we use DGNN to analyze the temporal dependencies of execution traces represented in SCPG, which can capture both spatial and temporal patterns of parallel programs. We refer to our approach as a Spatial-Temporal Dynamic Graph Neural Network (ST-DGNN). With ST-DGNN capturing the spatial and temporal patterns along the timeline, we further combine VAE to analyze the execution traces to identify performance anomalies caused by various performance inefficiencies in a unified way. Moreover, we propose an aggregation-based root cause diagnosis to pinpoint the most significant time regions or code regions where performance inefficiencies occur. With the guidance of root cause diagnosis, the developers can effectively optimize the problematic codes accordingly.

To materialize the above methods, we implement *STAD*,¹ a performance inefficiency detection tool for parallel programs. *STAD* captures the spatial and temporal patterns of parallel programs by analyzing the trace data to identify performance inefficiencies. Specifically, *STAD* intercepts the MPI function calls to collect the trace data and construct SCPG to capture the spatial patterns between processes. Then, *STAD* adopts the ST-DGNN model to capture the evolving program behaviors along the timeline to identify performance inefficiencies. Additionally, *STAD* also diagnoses the root causes of performance inefficiencies by aggregating the call path of anomalous functions. With the guidance provided by *STAD*, the developers can better optimize the target program accordingly. The advantage of *STAD*

compared to existing performance analysis tools comes from three folds: **1) a broader detection scope of performance inefficiencies** that relieves the burden of the developers to switching among different tools, **2) independent from expert knowledge for detection** that eases its adoption for non-expert developers, and **3) none reliance on labeled datasets for detection model training** that eliminates the overhead of dataset engineering and also improves detection accuracy.

Specifically, this paper makes the following contributions:

- We propose a novel spatial communication pattern graph (SCPG) to reorganize the MPI trace of parallel programs and capture the spatial patterns between processes in parallel programs. We further construct enclosing subgraphs of SCPG along the timeline to capture the evolving temporal patterns in the trace.
- We propose a spatial-temporal dynamic graph neural network (ST-DGNN) to represent the evolving spatial and temporal patterns of SCPG and detect the performance inefficiencies of parallel programs in a unified way. With ST-DGNN, we can detect performance anomalies caused by various performance inefficiencies.
- We propose an aggregation-based root cause diagnosis for detected performance inefficiencies to pinpoint the most significant time regions or code regions where the performance inefficiencies occur. We also provide both heatmap and tree views of diagnosis results to assist developers in understanding the root causes of performance inefficiencies.
- We implement *STAD*, a performance inefficiency detection tool for parallel programs, and evaluate *STAD* on several representative real-world parallel programs. Compared to the state-of-the-art performance analysis tools, we demonstrate that *STAD* can better identify several performance inefficiencies simultaneously with accurate root cause diagnosis and intuitive visualization reports.

II. OVERVIEW

STAD identifies the performance anomalies with root cause diagnosis by jointly considering the evolving spatial and temporal patterns. *STAD* offers a complete workflow for comprehensive performance analysis, which can directly work on production-level binary programs without recompilation. Overall, *STAD* trains a model using the trace of the target program's single execution and performs automated performance analysis. The design overview and workflow of *STAD* are shown in Fig. 1, which consists of four detection phases as follows:

Phase 1: MPI Trace Collection - *STAD* provides a runtime library to intercept the MPI function calls and collects the trace of parallel programs. For communication pattern and performance anomaly detection, the *STAD* library can obtain required communication parameters and performance counters from performance monitoring unit (PMU) with execution traces for further SCPG sequence construction and performance analysis.

Phase 2: SCPG Sequence Construction - *STAD* determines the duration and interval for partitioning the target traces into time slices in order to capture the spatial and temporal patterns.

¹STAD stands for the *Spatial-Temporal Anomaly analysis and Diagnosis*.

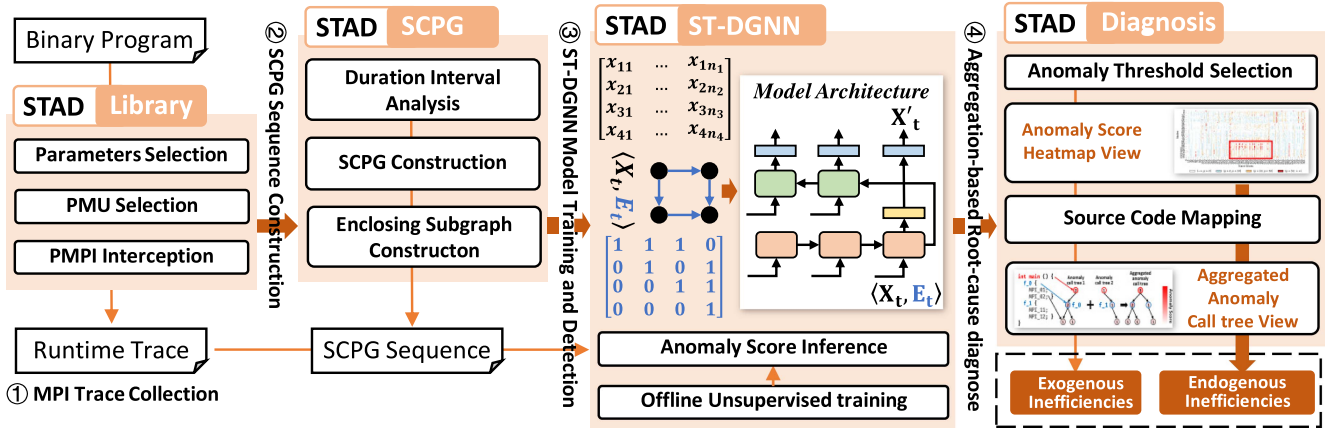


Fig. 1. The design overview and workflow of STAD.

Specifically, for traces within each time slice, *STAD* constructs the spatial communication pattern graph (SCPG) to capture the running node performance and spatial relationships via performance counters and function parameters, respectively. We further construct enclosing subgraphs of SCPG to improve detection accuracy. By using sliding windows along the timeline, *STAD* further constructs SCPG sequence to represent the temporal patterns of the target traces.

Phase 3: ST-DGNN Model Training and Detection - *STAD* first performs embedding on each SCPG to obtain a series of continuous graph vectors corresponding to the SCPG, which include both node vector and adjacency matrices. Then *STAD* applies offline unsupervised training of the proposed spatial-temporal dynamic graph neural network (ST-DGNN) model to capture the evolving spatial and temporal patterns within SCPG sequence, enabling detection of performance anomalies of parallel programs caused by different performance inefficiencies in a unified way. After training, *STAD* employs the ST-DGNN model to obtain the anomaly scores of the nodes in each SCPG.

Phase 4: Aggregation-based Root Cause Diagnosis - *STAD* selects a proper threshold to identify the performance anomalies for further root cause diagnosis. Then, *STAD* provides an intuitive report using a heatmap based on the identified anomalous nodes to assess whether the execution exhibits endogenous and exogenous inefficiencies. Especially for endogenous inefficiencies, *STAD* extracts aggregated anomaly call tree view to further diagnose the root causes with source code attributions.

III. SPATIAL AND TEMPORAL TRACE ANALYSIS

A. MPI Trace Collection

STAD provides a runtime library to intercept the MPI function calls and collect performance data of the parallel programs. To obtain sufficient spatial and temporal patterns of the MPI communication, *STAD Library* records the function parameters and performance counters crucial to the spatial pattern and performance of each collected function event.

Parameter Selection - *STAD Library* has integrated the collection of communication parameters for the commonly used MPI functions, such as `MPI_Send`, `MPI_Recv`, etc. *STAD* collects

TABLE I
THE PERFORMANCE COUNTERS SELECTED IN *STAD*

PMU	Description
BR_NTK	Conditional branch instructions not taken
LD_INS	Load instructions
L2_ICR	Level 2 instruction cache reads
BR_PRC	Conditional branch instructions correctly predicted
BR_MSP	Conditional branch instructions mispredicted
RES_STL	Cycles stalled on any resource
SR_INS	Store instructions
TOT_INS	Instructions completed

communication parameters of MPI communications to reconstruct the spatial communication patterns of parallel programs, such as the source/destination rank, the communication volume, and the communicator of specific communications.

Performance Counter Selection - Existing works have demonstrated the effectiveness of using performance counters to model the power consumption [31], [32], energy consumption [33], and performance [34], [35], [36] of parallel programs. *STAD* provides the capability to collect performance counters using the PAPI interface, allowing domain experts to flexibly select the data they wish to collect. Drawing inspiration from existing works, we selected eight parameters—`PAPI_BR_NTK`, `PAPI_LD_INS`, `PAPI_L2_ICR`, `PAPI_BR_PRC`, `PAPI_BR_MSP`, `PAPI_RES_STL`, `PAPI_SR_INS`, `PAPT_TOT_INS`, and `PAPI_L2_DCR`—for users to choose from, depending on the availability of these counters on the platform. The descriptions of these performance counters are detailed in Table I.

For users without domain-specific expertise, we recommend using the four performance counters `PAPI_LD_INS`, `PAPI_L2_ICR`, `PAPI_BR_PRC`, and `PAPT_TOT_INS`, which is the default configurations of *STAD*. They encompass instruction data related to computation, memory access, branch instructions, and load and store operations, providing a comprehensive representation of performance characteristics during program execution. These counters serve as critical inputs for model learning. This configuration is employed in all experiments conducted in this paper and has been proven to yield favorable results (detailed in Section IV).

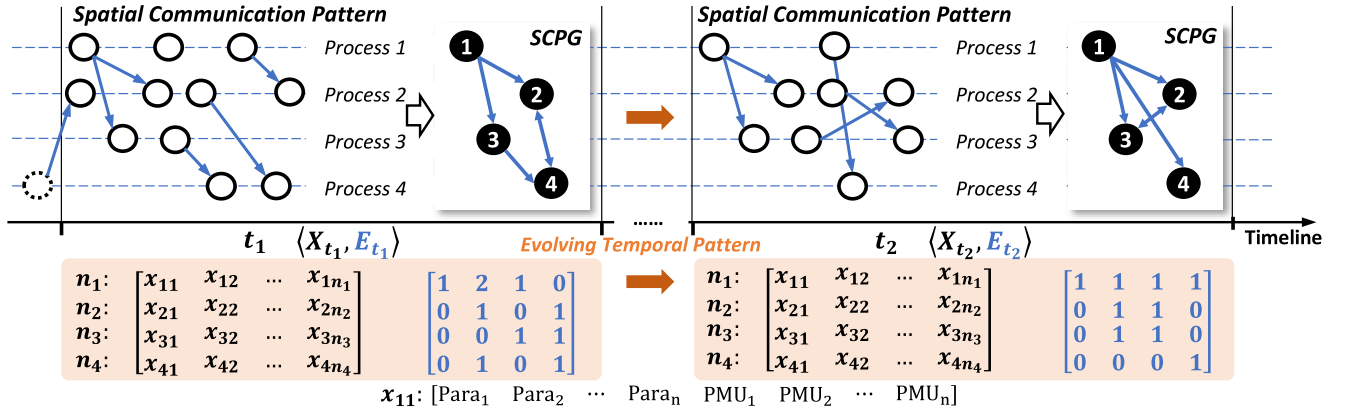


Fig. 2. An illustrative example of SCPG construction. Each black circle represents a MPI event record, and each blue edge with a arrow represents the communication between two MPI events.

Implementation - STAD Library intercepts all MPI functions through the MPI standard profiling interface (PMPI) and collects the performance counters through PAPI [37]. Specifically, it inserts data collection functions before and after each MPI function call to collect the communication parameters and performance counters. The collected data is then stored in trace files for further analysis.

B. SCPG Sequence Construction

STAD first analyzes the collected trace and partitions the trace into a series of trace slices along the timeline, then constructs the Spatial Communication Pattern Graph (SCPG) and SCPG sequence to represent the evolving spatial and temporal pattern of the trace within a slice. The detailed process to construct SCPG and SCPG sequence is described as follows.

SCPG - To characterize and aggregate the spatial communication patterns of the MPI communication, we propose a spatial communication pattern graph (SCPG) to reorganize the collected performance data of a trace slice. SCPG $\langle X_t, E_t \rangle$ is a directed graph that represents the spatial communication patterns of the MPI communications. The nodes X_t in the SCPG represent the node vectors of SCPG in time slice t , with each node consisting of an MPI function event sequence within the trace slice. For each MPI function event x_{ij} , the communication parameters and performance counters from PMU are stored in the node to represent the spatial patterns. The adjacency matrix E_t in the SCPG represent the communication patterns between the processes, and the weights of adjacency matrix are utilized to represent the communication direction and frequency.

Fig. 2 provides an example of the construction process of SCPG. Specifically, there are four processes in two trace slices t_1 and t_2 along the timeline, and *STAD* constructs an SCPG for each trace slice. For example, the nodes X_{t_1} of SCPG in slice t_1 constitute a set of nodes $[n_1, n_2, n_3, n_4]$, where each node n_i represents a sequence of MPI functions $[x_{i1}, x_{i2}, x_{i3}, \dots, x_{im_i}]$ with m_i MPI function events in the trace slice of process i . As shown in Fig. 2, the function sequence of n_1 in the SCPG $\langle X_{t_1}, E_{t_1} \rangle$ is $[x_{11}, x_{12}, x_{13}]$, whereas in the SCPG $\langle X_{t_2}, E_{t_2} \rangle$, the function sequence of n_1 is $[x_{11}, x_{12}]$. For each MPI function

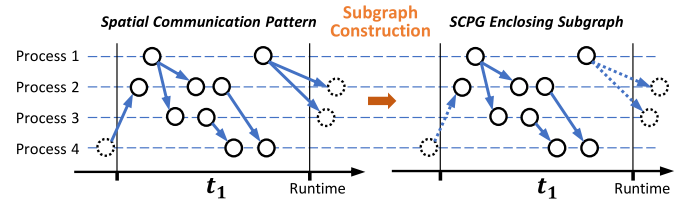


Fig. 3. The construction of SCPG enclosing subgraph.

event, the node leverages selected communication parameters (a.k.a., Para in Fig. 2) obtained by PMPI and performance counters (a.k.a., PMU in Fig. 2) obtained by PAPI to represent the spatial communication pattern, which is then used in performance anomaly detection.

For the edges within an SCPG, we adopt an adjacency matrix E_t to depict both the direction of communication. The adjacency matrix is structured as an $N \times N$ matrix, where N denotes the total number of nodes within the SCPG. Each element e_{ij} of the matrix indicates the communication from node i to node j . For example, in the SCPG $\langle X_{t_1}, E_{t_1} \rangle$, the elements e_{14} and e_{41} are both 0, indicating the absence of communication between *Process 1* and *Process 4*. In contrast, within the SCPG represented by $\langle X_{t_2}, E_{t_2} \rangle$, the element e_{14} is 1 and e_{41} is 0, denoting a unidirectional communication flow from *Process 1* to *Process 4*.

Enclosing Subgraph Construction - To improve the accuracy of SCPG for representing the communication pattern and performance characteristics within a trace slice, *STAD* further extracts the enclosing subgraph for SCPG. Specifically, the enclosing subgraph is a subgraph of SCPG that contains all the nodes and almost all the edges of SCPG, excluding the communications that do not belong to the trace slice completely. For example, as shown in the left part of Fig. 3, there are three functions in *Process 2*, *Process 3*, and *Process 4* out of slice t_1 communicate with the functions in the t_1 (dashed circles). Generally, these three functions will not be included in SCPG due to time slicing, whereas the communication edges are included, which may lead to incorrect analysis results.

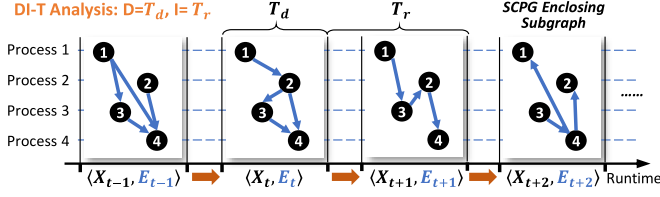


Fig. 4. Illustration of duration-interval analysis results.

Therefore, *STAD* discards the non-enclosing edges (dashed arrows) and the functions related to them (dashed circles) to construct the enclosing subgraph of SCPG. In the following sections, we refer to SCPG as the enclosing subgraph of SCPG for simplicity.

SCPG Sequence - After enclosing subgraph construction, the enclosed subgraphs are further organized into SCPG sequences to capture the temporal relationships of the trace slices. Specifically, *STAD* constructs SCPG enclosing subgraph sequences by sliding windows of constructed SCPG enclosing subgraphs along the timeline with a fixed sequence length and stride.

Duration-interval Analysis - For SCPG sequence construction, the *duration* of each time slice and *interval* between slices determine the number of MPI functions within a trace slice and the total number of trace slices. Specifically, Fig. 4 illustrates the usage of *duration* (T_d) and *interval* (T_r) configuration in SCPG sequence construction. When the T_d is greater than the T_r , the adjacent SCPGs can overlap to some extent in order to ensure all SCPGs collectively cover the entire program by introducing some redundant nodes among overlapped slices. Therefore, higher T_d leads to better semantic richness with less constructed SCPGs, whereas lower T_r leads to more SCPGs and less discarded non-enclosing edges with more redundancies. Since the detection accuracy will be affected by the richness of semantic features within a single SCPG as well as the number of SCPG sequences for stable unsupervised learning, the trade-off between the *duration* and *interval* should be carefully considered. Based on our empirical study, we choose both $T_d = T_r = 100$ ms as default in *STAD*, which achieves satisfactory detection accuracy in our experiments.

C. ST-DGNN

To identify the performance anomalies within a single execution trace in a unified way, *STAD* employs the dynamic graph neural network and the VAE stochastic model to learn the distribution of the latent variables instead of the latent variables themselves, which is referred as Spatial-Temporal Dynamic Graph Neural Network (ST-DGNN). Fig. 5 illustrates the overall workflow of ST-DGNN. Specifically, for each constructed SCPG enclosing subgraph, *STAD* first embeds X_t into fixed length feature vectors and constructs SCPG sequence with the embedded feature vectors as the input of ST-DGNN. During the model training stage, the ST-DGNN model is trained with the preprocessed dataset from the trace data. After the model is trained, a scoring function is used to compute the anomaly scores for each node in the SCPG to detect the performance

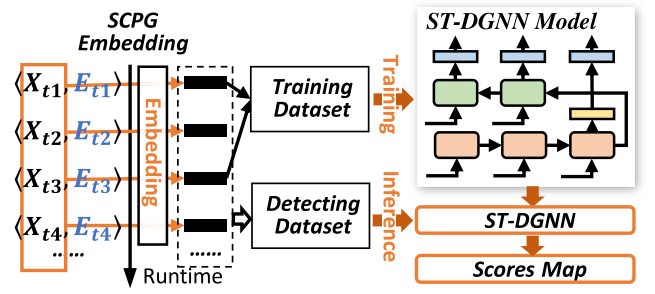


Fig. 5. The overall workflow of ST-DGNN.

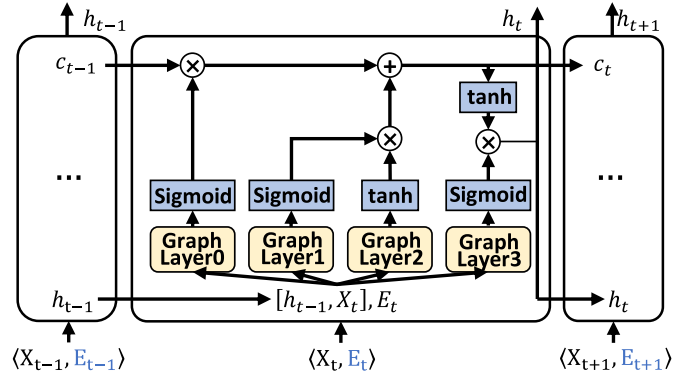


Fig. 6. The design structure of ST-LSTM.

anomalies, where a higher score indicates more likely the node is an anomaly.

Dynamic Graph Preprocessing - Due to the varying MPI events contained within each node, which include string-based attributes such as MPI event names, we preprocess the SCPGs into dynamic graphs with node embeddings to enable efficient model training and inference. Specifically, for each SCPG, we first encode the string-based node attributes into a numerical feature vector by applying the Word2Vec model [38] on the entire node attributions for nodes in all SCPGs. To better represent the evolving communication patterns, we use a dynamic edge adjacency matrix for each SCPG to construct a *dynamic graph topology*, as shown in Fig. 2. In the following sections, the input node refers to the preprocessed node feature embeddings from the Word2Vec model. Each element of the adjacency matrix, referred to as the weight, represents the communication frequency between nodes.

ST-LSTM - GraphLSTM [27] is previously designed to detect performance anomalies using profiling data for cloud systems with a static graph topology. Inspired by it, we propose ST-LSTM as a basic building block of ST-DGNN to support dynamic graph topology. ST-LSTM replaces the fully connected layers in LSTM with graph neural networks, such as GCN (Graph Convolutional Networks) [39] and GAT (Graph Attention Networks) [40], as shown in Fig. 6. The input of the ST-LSTM cell is the node vector X_t , the hidden state h_{t-1} , and a topology denoted by an input edge set array E_t . When a new input $\langle X_t, E_t \rangle$ from a SCPG sequence X arrives at time t , ST-LSTM updates the cell state c_t and calculates the new hidden

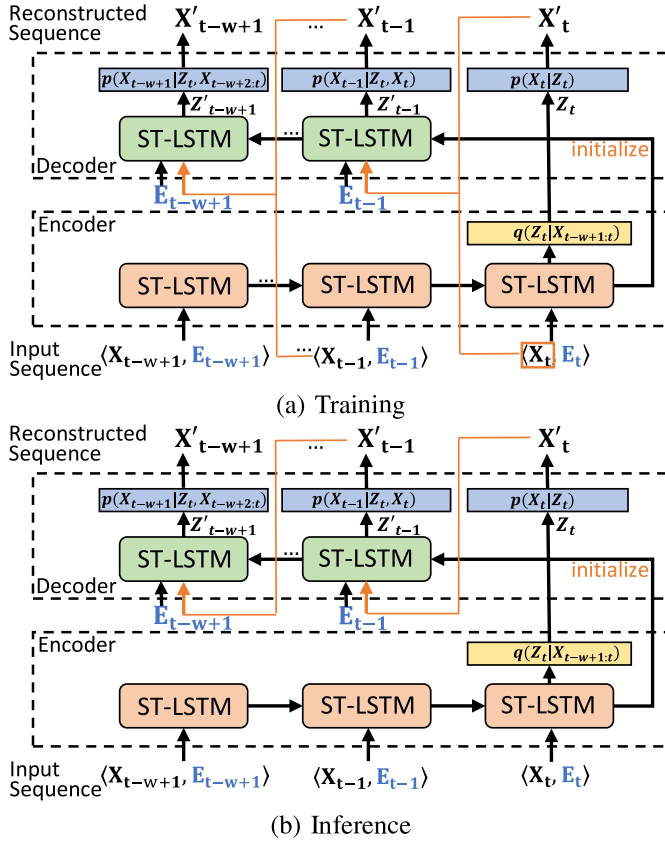


Fig. 7. The training and inference stage of ST-DGNN.

state h_t based on h_{t-1} and c_{t-1} . The computations conducted by ST-LSTM can be formulated as Equation 1, where f_t , i_t , g_t , c_t , o_t , and h_t is the forget gate, input gate, input modulation gate, cell state, output gate, and hidden state of the ST-LSTM cell, respectively. The $\mathcal{G}$ denotes the graph neural network operator, which can be GCN or GAT, and the $*$ denotes the Hadamard product.

$$\begin{aligned}
 f_t &= \text{sigmoid}(\mathbf{W}_f * \mathcal{G}([h_{t-1}, X_t], E_t) + \mathbf{b}_f) \\
 i_t &= \text{sigmoid}(\mathbf{W}_i * \mathcal{G}([h_{t-1}, X_t], E_t) + \mathbf{b}_i) \\
 g_t &= \tanh(\mathbf{W}_g * \mathcal{G}([h_{t-1}, X_t], E_t) + \mathbf{b}_g) \\
 c_t &= f_t * c_{t-1} + i_t * g_t \\
 o_t &= \text{sigmoid}(\mathbf{W}_o * \mathcal{G}([h_{t-1}, X_t], E_t) + \mathbf{b}_o) \\
 h_t &= o_t * \tanh(c_t)
 \end{aligned} \quad (1)$$

ST-DGNN Architecture - The training and inference of ST-DGNN are depicted in Fig. 7(a) and (b), respectively. Specifically, ST-DGNN is a stochastic sequence to sequence autoencoder [41], designed to capture evolving spatial and temporal patterns by processing the entire input sequence and reconstructing it in the decoder. Instead of directly learning the latent variables, the model learns the distribution of these latent variables. We use W to denote the length of the input SCPG sequence, $X_{t_0:t}$ to denote the input nodes of the SCPG sequence from time t_0 to t , X_t to denote the input nodes of SCPG at

time t , and z to denote their corresponding latent variables. The encoder of ST-DGNN computes the distribution $q(z_t|X_{t_0:t})$ and estimates the mean $\mu(X_{t_0:t})$ and standard deviation $\sigma(X_{t_0:t})$ of $q(z_t|X_{t_0:t}) = \mathcal{N}(\mu(X_{t_0:t}), \sigma(X_{t_0:t}))$ after processing the entire SCPG sequence. During training, we adopt a scheduled sampling method [1], [42], where the decoder's input at each time step is drawn from the original input and reconstructed sequence with probability λ and $1 - \lambda$, respectively (starting at 1). The states of the decoder are initialized by the final states of the encoder and the input at each time step is always derived from the preceding reconstruction. Specifically, for an SCPG $\langle X_{t-1}, E_{t-1} \rangle$, the input is the mean of the distribution $p(X_t|z_t) = \mathcal{N}(\mu(z_t), \sigma(z_t))$.

Offline Model Training - ST-DGNN is trained to obtain the neural network parameter ϕ and θ by maximizing the variational lower bound $\mathcal{L}(\theta, \phi; X_{t_0:t})$ on the marginal likelihood. Given that a SCPG node sequence $X_{t_0:t}$ within a time window, the objective function is formulated as (2):

$$\begin{aligned}
 \mathcal{L}(\theta, \phi, X_{t_0:t}) &= -D_{KL}(q_\phi(z_t|X_{t_0:t})||p_\theta(z_t)) \\
 &\quad + E_{q_\phi(z|X_{t_0:t})}[\log(p_\theta(X_{t_0:t}|z_t))] \\
 &= -D_{KL}(q_\phi(z_t|X_{t_0:t})||\mathcal{N}(0, 1)) \\
 &\quad + \frac{1}{L} \sum_{l=1}^L \sum_{j=t_0}^t \log(p_\theta(X_j|z_t^{(l)}, X_{j+i:t})) \quad (2)
 \end{aligned}$$

The objective function [43] is the sum of the negative of KL divergence D_{KL} between the posterior distribution $q_\phi(z_t|X_{t_0:t})$ and the prior distribution $p_\theta(z_t)$, and the reconstruction probability E_{q_ϕ} of the input data $X_{t_0:t}$. L denotes the expected sampling number, which is set to 1 as reported in [43]. The SCPG sequence serves as the input to the model, and the stopping criterion for training is either model convergence or when the loss no longer decreases after 60 epochs. It is important to note that the performance of the model is not influenced by the final value of the loss function.

Computing Anomaly Score - Generally, we can calculate the reconstruction error $-E_{q_\phi(z|X_{t_0:t})}(\log(p_\theta(X_t|z_t)))$ at trace slice t and use it as the anomaly score S_t to detect the performance anomalies. Considering the latency and correlation of performance anomalies within a program, as well as the requirement to individually identify every node i within each SCPG, the final anomaly score is expressed as (3).

$$S_t = -\frac{1}{L * D} \sum_{d=0}^{D-1} \sum_{l=1}^L \log(p_\theta(X_t^i|z_{t+d}^{(l)}, X_{t+1:t+d})) \quad (3)$$

where X_t^i denotes the i -th node in X_t , and D denotes the length of the time window to calculate the anomaly score.

D. Aggregation-Based Root Cause Diagnosis

The anomaly scores S_t are computed in a manner that is positively correlated with reconstruction loss. Therefore, a higher anomaly score indicates a higher likelihood that the node is a performance anomaly. Once the anomaly scores are computed, STAD employs an aggregation-based root cause diagnosis to

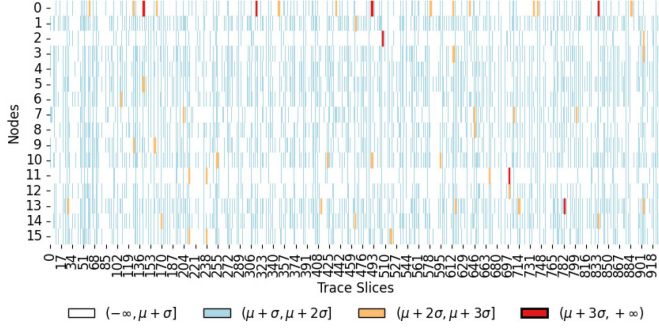


Fig. 8. The heatmap visualization of a normal program execution without notable inefficiency.

further analyze anomaly score distributions and the aggregation feature of code anomalies. *STAD* provides both heatmap and tree view to aid developers in distinguishing between endogenous and exogenous inefficiencies for a particular execution with root cause diagnosis.

Anomaly Threshold Selection - To distinguish the normal nodes from the abnormal nodes, *STAD* employs an anomaly threshold τ to classify the nodes as normal and abnormal. Since the distribution of anomaly scores usually follows a normal distribution with long tails, we leverage the mean and standard deviation of the anomaly scores to calculate the threshold τ as $\tau(\alpha) = \mu + \alpha * \sigma$, where μ and σ are the mean and standard deviation of the anomaly scores, respectively, and α is a hyper-parameter to control the threshold. According to the property of normal distribution, we choose $\alpha = 1, 2, 3$ to calculate three successive anomaly thresholds to assess the confidence and severity of anomalies. In general, we treat the nodes with anomaly scores larger than $\tau(1)$ as potential performance anomalies.

Exogenous Inefficiency Diagnosis - The exogenous inefficiency diagnosis is to identify the performance anomalies caused by the running environments, such as network congestion, hardware variance, and system noise. *STAD* employs a visualization approach to display the distribution of anomaly scores, providing an anomaly heatmap view to identify the endogenous inefficiency intuitively. Since *STAD* re-evaluates the entire SCPG sequence and computes anomaly scores after model training, the anomaly scores for all nodes are naturally normalized by ST-DGNN. Thus, we can directly utilize the anomaly scores of each node across all SCPGs to generate a heatmap, which visually facilitates the identification of endogenous inefficiencies. In the heatmap, the horizontal axis represents the SCPGs (along the timeline) and the vertical axis represents the node index (e.g., MPI ranks).

As shown in Fig. 8, this is a heatmap of anomaly scores generated using a trace from a normal program execution without notable inefficiency. The dominant color of the heatmap is similar to the lower portion of the anomaly score color bar at the bottom (e.g., white or light blue), indicating that most nodes are normal with relatively low anomaly scores. When significant performance anomalies occur, particularly those caused by exogenous inefficiencies, dark-colored blocks (e.g., orange or red) aggregate at the node level. These blocks indicate that the

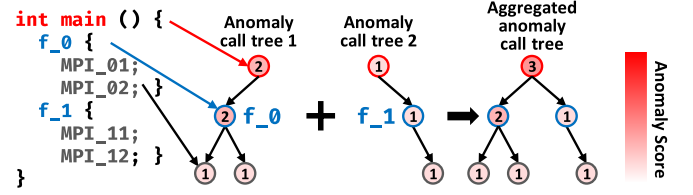


Fig. 9. An example to construct an aggregated anomaly call tree.

TABLE II
THE HARDWARE AND SOFTWARE SPECIFICATIONS

Server	Tracing	Analyzing
Processor	Intel Golden 6240 @2.60GHz	Intel Xeon Gold 6330 @2.00GHz
Nodes	32	1
Cores	36	56
Memory	384 GB	256 GB
GPU	-	Tesla V100
Software	Openmpi 4, gcc 9.4.0 -O3	
CUDA	-	11.8
PyTorch	-	2.1.0

performance inefficiencies can be attributed to the anomalies in hardware associated with corresponding nodes during program execution.

Endogenous Inefficiency Diagnosis - Endogenous inefficiency diagnosis aims to identify performance anomalies originating from the code itself. *STAD* offers an intuitive and aggregated anomaly call tree view that facilitates the identification of endogenous inefficiencies and their root causes. Fig. 9 provides an example to construct an aggregated anomaly call tree. Specifically, the root of the aggregated anomaly call tree is the main function of the target program, while the deepest leaf nodes of the tree correspond to the MPI functions. To construct the aggregation tree, *STAD* first filters nodes with anomaly scores exceeding the threshold $\tau(1)$. For each anomaly node, *STAD* extracts functions along with their corresponding backtrace and source code attributions. The anomaly call tree is then built based on these functions, with each node's anomaly score reflecting the severity of the anomalies. Finally, *STAD* aggregates the individual anomaly call trees to form a comprehensive anomaly-aggregated call tree. The scores in the aggregated tree are the sums of the corresponding node score from each individual anomaly call tree.

IV. EVALUATION

A. Experimental Setup

The hardware and software specifications are summarized in Table II. We deploy *STAD*² on a server equipped with dual Intel Xeon Gold 6330 CPUs and a Tesla V100 GPU (referred to as *deployment server*). For the anomaly detection dataset, we collect data from both the deployment server and a home-built cluster with 32 nodes. Each node in the cluster is configured with dual Intel Gold 6240 CPUs, and 384 GB of memory, and the nodes are interconnected via an IB EDR switch. We compile

²<https://github.com/Synlvejo/STAD>

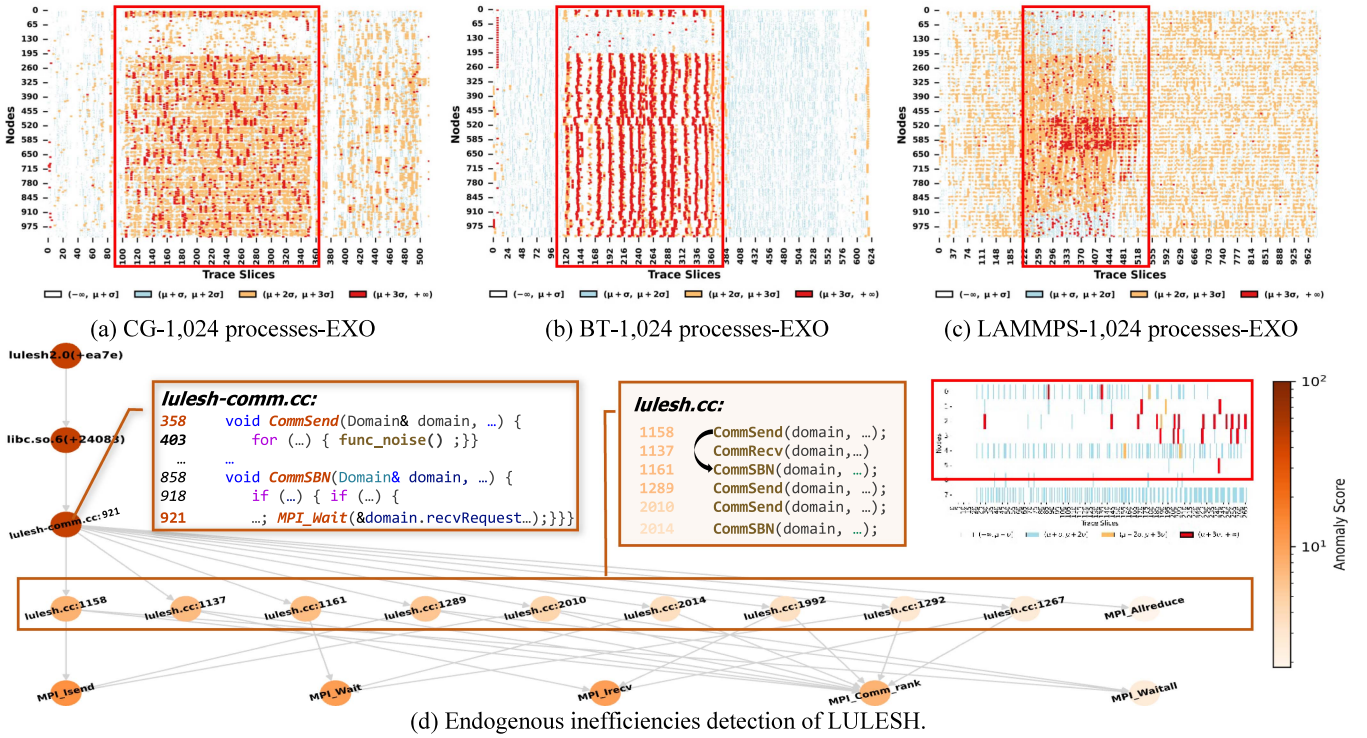


Fig. 10. The detection of exogenous and endogenous inefficiencies.

all programs with GCC 9.4.0 and OpenMPI 4.0.4. We develop ST-DGNN with PyTorch 2.1.0 compiled with CUDA 11.8. We evaluate the effectiveness of *STAD* using a set of representative parallel programs including 1) *NPB benchmarks* [44], which are widely utilized in the HPC community; 2) *LAMMPS* [45], a classical molecular dynamics simulation software extensively used across various fields; and 3) *LULESH*, a real-world application designed to simulate the behavior of hydrodynamic systems under extreme conditions. For evaluation, we use a heavy workload program to simulate system noise for exogenous inefficiency injection. For hyperparameters in *STAD*, we set the duration T_d to 100 micro-seconds and sliding window size to 30 to construct the SCPG sequence. This configuration is also designed to be directly used by non-experts in the field, and experimental results have demonstrated its effectiveness. For ST-DGNN, we set the learning rate to 0.001 and stop the model training when the loss function stagnates among 60 epochs. For all experiments, we select `PAPI_LD_INS`, `PAPI_L2_ICR`, `PAPI_BR_PRC`, and `PAPT_TOT_INS` as the performance counters to characterize the program performance.

B. Exogenous Inefficiency Diagnosis

To evaluate the effectiveness of *STAD* in diagnosing exogenous inefficiencies, we conduct experiments using the *NPB* benchmark (D CLASS input) and *LAMMPS* with 1,024 processes. We inject an exogenous inefficiency lasting 20 seconds into the program execution by running a heavy workload program, starting 15 seconds after execution commenced. Performance inefficiencies are then detected with the proposed

STAD analysis workflow. As shown in Fig. 10(a)~(c), we demonstrate the resulting heatmap views of the detected performance anomalies for each evaluated program. Points exhibiting distinct aggregation features detected by *STAD* are highlighted with red boxes. Specifically, as shown in Fig. 10(a), the red anomaly points align with the higher-scoring orange regions, covering the anomaly injection period (time slices 110 to 360). In Fig. 10(b), the anomalous regions are observed in the middle range (120 to 370). Fig. 10(c) shows that red, orange, and blue anomaly points are concentrated between the 220-th and 470-th time slices, approximately 25 seconds. These results align with our injection methodology and demonstrate that *STAD* effectively identifies exogenous inefficiencies occurring during software execution across multiple scales.

Users can leverage the anomaly heatmap view to determine whether the program execution is affected by exogenous inefficiencies and identify straggler nodes. By contacting system administrators or excluding these straggler nodes, users can enhance application performance. Additionally, the heatmap view can assist in detecting irregular anomalous behaviors, indicating that further endogenous inefficiency diagnosis is necessary.

C. Endogenous Inefficiency Diagnosis

For the effectiveness of *STAD* in endogenous inefficiencies diagnosis, we use *LULESH* with injected code inefficiencies and evaluate it with *STAD* with 8 processes on the development server. Specifically, *Comm_Send* is a critical communication function to broadcast data among processes. For validation, we modify *LULESH* by adding intensive computations for the first

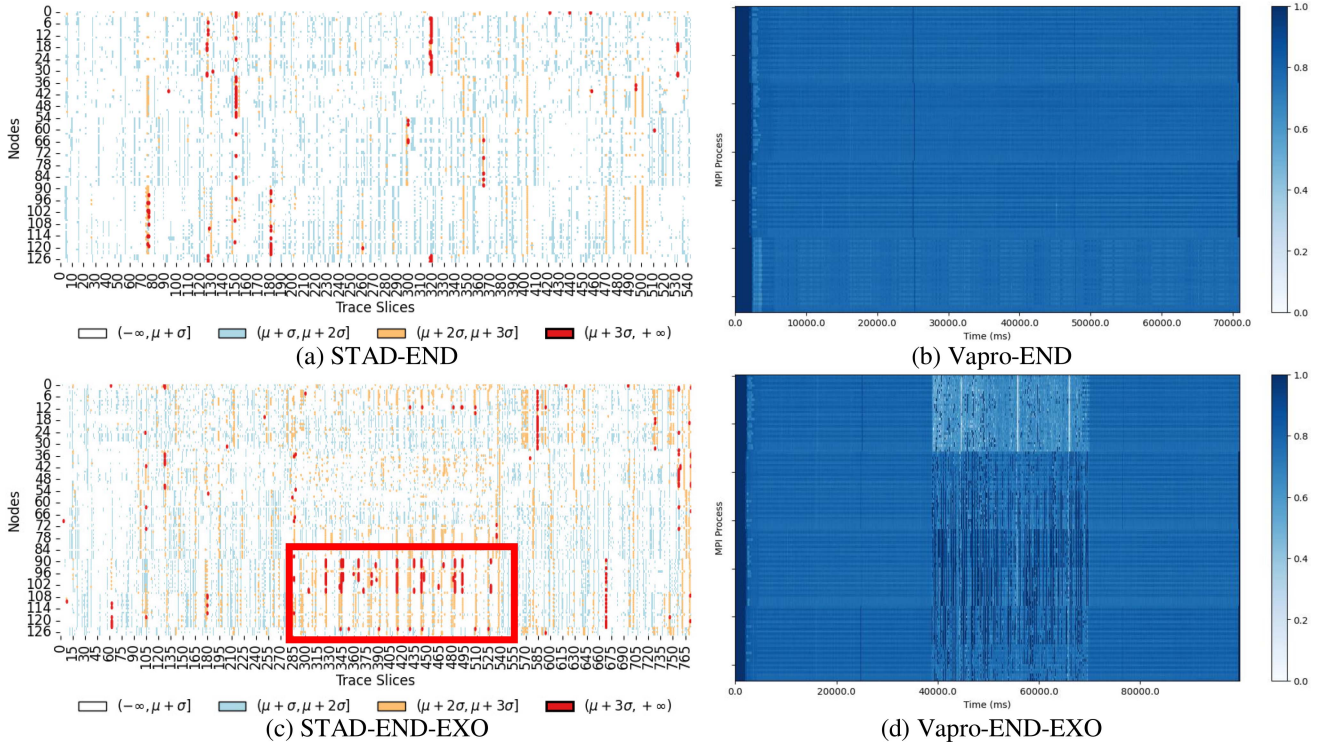


Fig. 11. LAMMPS running 128 processes with STAD and Vapro.

four MPI processes in the *Comm_Send* function at *lulesh-comm.cc:403*, injecting as inefficient communication routines. The detected endogenous inefficiencies are illustrated in Fig. 10(d). Firstly, we leverage an anomaly heatmap view to reveal numerous irregular patterns in the first four processes. Consequently, we apply the endogenous inefficiencies diagnosis to the red anomalous points. Upon constructing the aggregated anomaly call tree, we identify *lulesh-comm.cc:921* as the root cause of these anomalies. Specifically, this root cause corresponds to the *MPI_wait* function invoked by the *CommSBN* function. Further investigation into the *MPI_Wait* function allows us to isolate the code associated with the most anomalous nodes in *lulesh.cc*. We observe that the most anomalous code locations consistently involve the *CommSend* function, along with its related functions *CommRecv* and *CommSBN*. Thus, these results validate the effectiveness of *STAD* in diagnosing endogenous inefficiencies.

D. Case Study: LAMMPS

To further assess the effectiveness of *STAD* in identifying both exogenous and endogenous inefficiencies, we conduct a practical end-to-end case study on LAMMPS with 128 processes and compared the results with the state-of-the-art performance analysis tool *Vapro* [20]. We collect traces from LAMMPS under two conditions: one with only endogenous inefficiencies and one with both endogenous and exogenous inefficiencies. We then evaluated and compared the effectiveness of *STAD* and *Vapro* in identifying these inefficiencies. The exogenous inefficiencies are injected with the heavy-workload program described in

Section IV-B. The endogenous inefficiencies refer to the load imbalance that has been reported in the previous work [12].

Analysis Workflow - We begin by tracing the execution of the LAMMPS program using the *STAD* library (Section III-A) and preprocessing the trace data to generate the SCPG sequence (Section III-B). The SCPG sequence is then vectorized via Word2Vec to serve as input for the ST-DGNN model. Next, we train the ST-DGNN model using these vectorized SCPG sequences. For inference, we utilize the SCPG sequence vectors to generate an anomaly heatmap (Section III-C), which helps identify exogenous inefficiencies. If irregular anomalies are detected in the heatmap, we further construct an aggregated anomaly call tree based on the heatmap to pinpoint endogenous inefficiencies (Section III-D). Finally, we apply the corresponding optimization strategies to LAMMPS. The details of trace collection, preprocessing, and model training are provided in Section IV-A.

Exogenous inefficiency detection - As shown in Fig. 11, we first present the anomaly heatmap view generated by *STAD* and *Vapro*. For the trace data with only endogenous inefficiencies, we observe that *STAD* can successfully detect the performance anomalous points (red color), as indicated in the Fig. 11(a), whereas the *Vapro* can not, as there is no white block in Fig. 11(b). *Vapro* detects exogenous inefficiencies by identifying the fixed-workload fragments with the same calling contexts (a.k.a., execution states) via function parameters (e.g., MPI communications) or PMU counters (e.g., *PAPI_TOT_INS* for computations), which can be further normalized to detect performance variances. As its detection approach is based on the insight that the fragments with the similar workload should

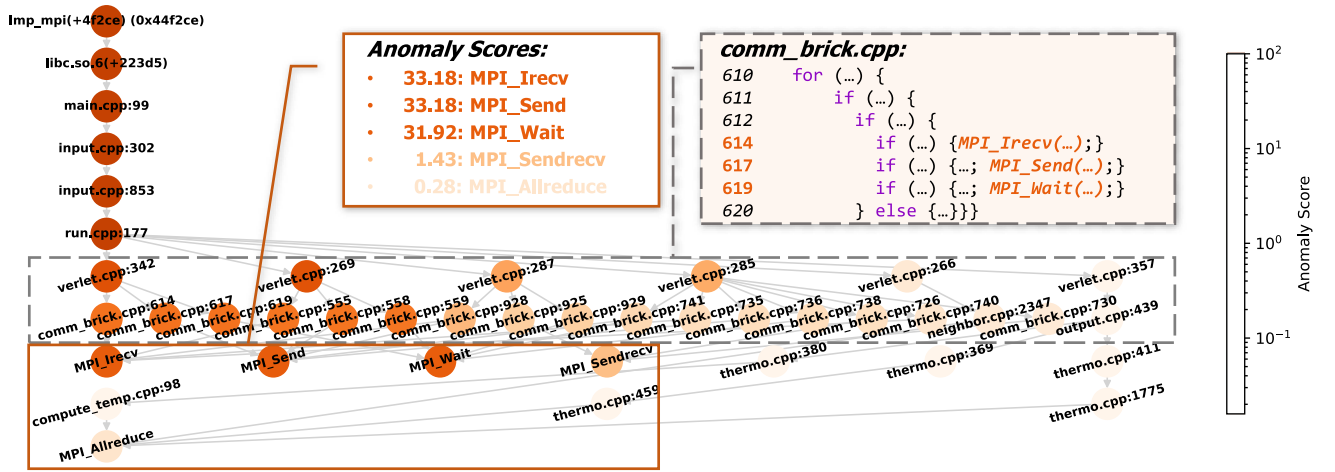


Fig. 12. The aggregated anomaly call tree of LAMMPS.

share the similar performance and thus the detected performance variance must be come from the hardware or system performance variances, which is not the case for endogenous inefficiencies. For the trace data with both endogenous and exogenous inefficiencies, *STAD* and *Vapro* can both detect the exogenous inefficiencies, as shown in Fig. 11(c) and (d). However, *STAD* can also detect the endogenous inefficiencies, as indicated by the red points out of the red block in Fig. 11(c), while the *Vapro* fails due to the lack of white block out of the anomalous time slices, as shown in Fig. 11(d). Consequently, the anomaly heatmap view generated by *STAD* reveals both types of inefficiencies with the anomalous regions clearly highlighted in red. In contrast, the anomaly heatmap view generated by *Vapro* is less detailed with the anomalous regions appearing as a single white block.

Endogenous inefficiency detection - Since the anomaly heatmap view indicates that there are performance inefficiencies not belong to the exogenous inefficiencies, we further analyze the endogenous inefficiencies using the output of the anomaly heatmap view. We filter the red anomalous points and construct the aggregated anomaly call tree to diagnose the root causes of these inefficiencies. As shown in Fig. 12, we generate the aggregated anomaly call tree based on the anomalous points and aggregate the same node whose call path is the same. From anomaly scores of the tree, we find that the root cause of the endogenous inefficiencies is the *MPI_Irecv*, *MPI_Send*, and *MPI_Wait* functions in *comm_brick.cpp:614*, *comm_brick.cpp:617*, and *comm_brick.cpp:619*, respectively. Upon further analysis of the locations of these codes, we observe that all functions are situated within the same loop at line *comm_brick.cpp:610*, as shown in Fig. 12. This placement can result in an imbalance between the send and receive times across different processes during asynchronous communication, consequently leading to load imbalance issues.

Actionable Optimization - For the exogenous inefficiencies detected by *STAD*, we optimize program execution by excluding straggler nodes from computation (e.g., nodes 96 to 127). This

exclusion reduces execution time from 79 s to 55 s, achieving a speedup of $1.44\times$. For the endogenous inefficiencies identified by *STAD*, users can enhance execution efficiency by incorporating the balance function into the LAMMPS input file. Applying this balance optimization decreases execution time from 55s to 49s, resulting in a speedup of $1.12\times$. Overall, by following the actionable optimization guidance provided by *STAD*, we achieve a total speedup of $1.57\times$ in program execution.

The result of this case study demonstrates that *STAD* can effectively detect both exogenous and endogenous inefficiencies, providing a more intuitive and detailed analysis by analyzing one execution trace without multiple executions.

E. Ablation Study

To better understand the effectiveness of the major components such as *enclosing subgraph*, *dynamic topology*, and *performance counter*, we provide an ablation study of *STAD*. For better visualization, we present the results of the ablation study using the anomaly heatmap view generated by *STAD* without employing the anomaly score threshold and highlight the exogenous inefficiencies we inject with red blocks shown in Fig. 13. Specifically, we use the LU benchmark from the NPB suite to conduct the ablation study. Fig. 13(a) demonstrates an external detection scenario that excludes both the enclosing subgraph and the dynamic topology. The anomaly heatmap appears predominantly white, making it difficult to identify anomalous regions. Fig. 13(b) shows the effectiveness of enclosing subgraph techniques. Compared to Fig. 13(a), applying enclosing subgraph improves contrast while it is still challenging to identify the program trace pattern changes. In Fig. 13(c), incorporating performance counters into the model training enhances the clarity and prominence of pattern changes. Finally, Fig. 13(d) demonstrates that using all three methods together provides a distinctly visible representation of program pattern changes, facilitating more intuitive detection of performance anomalies.

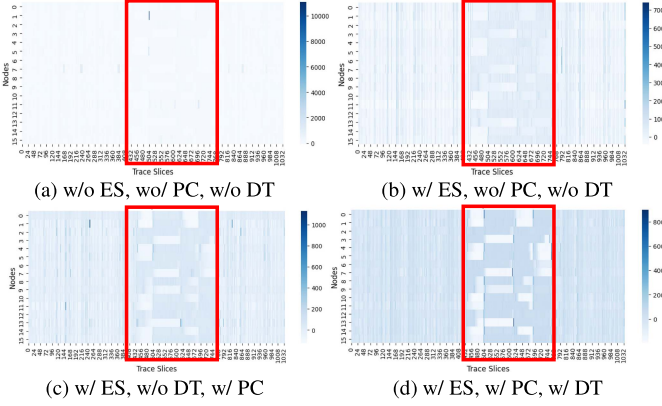


Fig. 13. The ablation study of STAD.

TABLE III
THE OVERHEAD OF STAD

Program	Scale	Storage (GB)	Training Time(min)	Inference Time(s)	Analysis Time(s)
LU	16	0.8	83	20	5
CG	1024	40	322	73	25
BT	1024	28	180	85	31
LAMMPS	1024	27	354	125	19
LULESH	8	0.1	15	10	5
LAMMPS	128	4.6	40	23	9

F. Overhead

The overhead of *STAD* comes from five major components: tracing, storage, ST-DGNN training, inference, and analysis overheads. Among these, tracing overhead increases with program scale and is dependent on the program, which is same with other state-of-the-art performance analysis tools [12], [46], [47], [48], [49], [50], [51]. *STAD* library incurs an average performance overhead of $1.22\times$ across all benchmark programs evaluated in our experiments at a scale of 1,024 processes. Storage overhead pertains to the amount of storage required for the performance trace collected by the *STAD* library. The analysis overhead refers to the cost associated with generating the anomaly heatmap view and the aggregated anomaly call tree within *STAD*. Detailed overheads for all evaluated programs are presented in Table III. Generally, the predominant overhead of *STAD* arises from the training of the ST-DGNN, with the total overhead increasing with the scale of the program, the duration of execution, and the volume of performance traces. During the anomaly detection phase, *STAD* incurs minimal overhead, typically completing within one minute. Furthermore, the training and analysis overhead can be further reduced through parallelization.

We further analyze the algorithmic complexity of *ST-DGNN* to provide an intuitive understanding of its computational overhead. The time complexity of *ST-DGNN* is formulated in (4), where L represents the number of layers, $|E|$ denotes the number of edges, d corresponds to the dimension of node feature vectors, T indicates the number of time steps, and N represents the total number of nodes.

$$O_{ST-DGNN} = O(T(L|E|d + LNd^2)) \quad (4)$$

V. DISCUSSION

Advantages of *STAD*: *STAD* addresses the narrow detection scope by employing an anomaly detection approach instead of heuristic analysis, which relies heavily on expert effort and empirical experience. To mitigate the dependence on expert knowledge, *STAD* introduces ST-DGNN and utilizes a machine learning-based approach to identify performance anomalies, attributing them to actionable exogenous and endogenous inefficiencies, such as straggler nodes and inefficient code lines. Furthermore, to address the limitation of lacking labeled datasets, *STAD* leverages an unsupervised learning method (VAE) to detect anomalies within a specific program execution rather than across all program types.

Limitations of *STAD*: *STAD* still has limitation in performance analysis. In practical deployment scenarios, for any performance analysis tools, the time required for detailed performance analysis scales linearly with the trace volume. For *STAD*, the overhead is inherently constrained by scale, but it can be further mitigated through parallelism in preprocessing and model training. Moreover, reducing the trace volume for large scale execution (e.g., trace compression) is also effective for ensuring practical performance analysis. In future work, we will explore techniques to enhance the efficiency of *STAD* to further broaden its applicability.

STAD Model: For different spatial-temporal neural networks, considering the training overhead, super parameters configuration, training resource need, and difference on preprocessing, we cannot offer a fair comparison with them. But we will also explore more popular and advanced machine learning models to improve the accuracy and efficiency of *STAD* in the future work.

VI. RELATED WORK

To identify the performance inefficiencies of parallel programs, researchers employ profiling and tracing tools [46], [47], [48], [50], [51], [52], [53], [54], [55] and apply corresponding performance analysis approaches [17], [20], [56], [57], [58], [59].

Endogenous Inefficiency Analysis - To detect endogenous inefficiencies buried deeply in software, researchers have proposed various heuristic-based methods. Scalana [12] has proposed a Program Structure Graph (PSG) to detect the root cause of scaling loss in parallel applications. Several value profilers [13], [15], [16] also have been proposed to detect memory inefficiencies via fine-grained value profiling. Perflow [17] designed a domain-specific programming framework to mitigate the burden of performance analysis implementation. STAT [11] detects the program hanging problem with automated stack trace analysis. There are also some machine learning-based approaches to detect endogenous inefficiencies. GRAPHSPY [10] has proposed a learning-aided approach to identify memory access inefficiencies. Puffin [14] presents a learning-based approach to identifying dead stores, silent loads, and silent stores. However, these works are limited to a specific type of endogenous inefficiency. To the best of our knowledge, *STAD* is the first to integrate an unsupervised learning approach into MPI trace analysis, offering a unified framework for detecting both endogenous and exogenous inefficiencies.

Exogenous Inefficiency Analysis - Exogenous inefficiency analysis is proposed to detect the performance inefficiencies that are caused by the system itself, such as operating system noise [23], CPU-related problems [2], and hardware-related issues [7], etc. Moreover, researchers have identified several additional sources of variability, including node-level performance failures [60], network congestion [5], hardware manufacturing process variations [3], [4], asynchronous checkpointing [25], [26], in-situ systems [61], [62], and system power caps [63]. Sinha et al. [7] clarify that the manufacturing variability of the hardware can cause up to 22% performance variance in HPC systems. Prodigy [6] detects performance anomalies at the node level by exploring the monitoring data. Vapro [20] and Vsensor [5] proposed a fixed workload detection identification to detect the performance variance caused by the exogenous inefficiencies. *STAD* leverages SCPG and ST-DGNN to capture the evolving temporal patterns in trace data, effectively expanding the detection scope beyond the limitations of existing tools.

Compared with these works, *STAD* focuses on detecting endogenous and exogenous inefficiencies in a unified way without much expert knowledge and helps users identify if there is a performance issue and diagnose the possible root causes of endogenous inefficiencies. *STAD* introduces the novel SCPG to characterize spatial performance while incorporating dynamic graph topology into DGNN for capturing evolving temporal patterns. By leveraging the ST-DGNN model, *STAD* enables the simultaneous identification of both endogenous and exogenous inefficiencies through an aggregation-based root cause diagnosis approach.

VII. CONCLUSION

In this paper, we propose *STAD*, a tool for spatial and temporal anomaly detection in parallel programs. *STAD* intercepts MPI function calls to collect performance traces and constructs spatial communication pattern graphs to capture the spatial relationships between processes. It then utilizes a spatial-temporal dynamic graph neural network model to analyze both spatial and temporal evolution along the timeline and detect performance anomalies. Additionally, *STAD* provides both anomaly heatmap and aggregated anomaly call tree views for root cause diagnosis. Our evaluation results show that *STAD* effectively identifies multiple performance anomalies caused by both exogenous and endogenous inefficiencies simultaneously with intuitive visualization reports and accurate root cause diagnosis.

REFERENCES

- [1] Z. He et al., "A spatiotemporal deep learning approach for unsupervised anomaly detection in cloud systems," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 34, no. 4, pp. 1705–1719, Apr. 2023.
- [2] A. Marathe, Y. Zhang, G. Blanks, N. Kumbhare, G. Abdulla, and B. Rountree, "An empirical survey of performance and energy efficiency variation on intel processors," in *Proc. 5th Int. Workshop Energy Efficient Supercomputing*, New York, NY, USA: Association for Computing Machinery, 2017, pp. 1–8, doi: [10.1145/3149412.3149421](https://doi.org/10.1145/3149412.3149421).
- [3] J. Eastep et al., "Global extensible open power manager: A vehicle for HPC community collaboration on co-designed energy management solutions," in *Proc. 32nd Int. Conf. High Perform. Comput.*, Frankfurt, Germany, 2017, pp. 394–412.
- [4] P. Sinha, A. Guliani, R. Jain, B. Tran, M. D. Sinclair, and S. Venkataraman, "Not all GPUs are created equal: Characterizing variability in large-scale, accelerator-rich systems," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2022, Art. no. 65.
- [5] X. Tang, J. Zhai, X. Qian, B. He, W. Xue, and W. Chen, "Vsensor: Leveraging fixed-workload snippets of programs for performance variance detection," in *Proc. 23rd ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2018, pp. 124–136.
- [6] B. Aksar et al., "Prodigy: Towards unsupervised anomaly detection in production HPC systems," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, New York, NY, USA: Association for Computing Machinery, 2023, Art. no. 26, doi: [10.1145/3581784.3607076](https://doi.org/10.1145/3581784.3607076).
- [7] P. Sinha, A. Guliani, R. Jain, B. Tran, M. D. Sinclair, and S. Venkataraman, "Not all GPUs are created equal: Characterizing variability in large-scale, accelerator-rich systems," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2022, pp. 1–15.
- [8] S. Moreno-Álvarez, J. M. Haut, M. E. Paoletti, J. A. Rico-Gallego, J. C. Díaz-Martín, and J. Plaza, "Training deep neural networks: A static load balancing approach," *J. Supercomput.*, vol. 76, no. 12, pp. 9739–9754, Dec. 2020, doi: [10.1007/s11227-020-03200-6](https://doi.org/10.1007/s11227-020-03200-6).
- [9] Y. He et al., "Understanding and mitigating hardware failures in deep learning training systems," in *Proc. 50th Annu. Int. Symp. Comput. Architecture*, New York, NY, USA: Association for Computing Machinery, 2023, doi: [10.1145/3579371.3589105](https://doi.org/10.1145/3579371.3589105).
- [10] Y. Guo, P. Li, Y. Luo, X. Wang, and Z. Wang, "Graphspy: Fused program semantic embedding through graph neural networks for memory efficiency," in *Proc. 58th ACM/IEEE Des. Automat. Conf.*, 2021, pp. 1045–1050.
- [11] 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 *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2007, pp. 1–10.
- [12] Y. Jin et al., "Automating scaling loss detection with graph analysis," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2020, pp. 1–14.
- [13] X. You, H. Yang, Z. Luan, D. Qian, and X. Liu, "Zerospy: Exploring software inefficiency with redundant zeros," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2020, pp. 1–14.
- [14] P. Li, Y. Guo, Y. Luo, X. Wang, Z. Wang, and X. Liu, "Graph neural networks based memory inefficiency detection using selective sampling," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2022, Art. no. 85.
- [15] X. You, H. Yang, K. Lei, Z. Luan, and D. Qian, "TrivialSpy: Identifying software triviality via fine-grained and dataflow-based value profiling," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2023, pp. 1–14.
- [16] X. You, H. Yang, K. Lei, Z. Luan, and D. Qian, "Vclinic: A portable and efficient framework for fine-grained value profilers," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2023, pp. 892–904.
- [17] Y. Jin, H. Wang, R. Zhong, C. Zhang, and J. Zhai, "PerFlow: A domain specific framework for automatic performance analysis of parallel applications," in *Proc. 27th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2022, pp. 177–191.
- [18] X. Meng, J. M. Anderson, J. Mellor-Crummey, M. W. Krentel, B. P. Miller, and S. Milaković, "Parallel binary code analysis," in *Proc. 26th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, New York, NY, USA: Association for Computing Machinery, 2021, pp. 76–89, doi: [10.1145/3437801.3441604](https://doi.org/10.1145/3437801.3441604).
- [19] K. Zhou, M. Krentel, and J. Mellor-Crummey, "A tool for top-down performance analysis of GPU-accelerated applications," in *Proc. 34th ACM Int. Conf. Supercomputing*, New York, NY, USA: Association for Computing Machinery, 2020, pp. 415–416, doi: [10.1145/3332466.3374534](https://doi.org/10.1145/3332466.3374534).
- [20] L. Zheng et al., "Vapro: Performance variance detection and diagnosis for production-run parallel applications," in *Proc. 27th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2022, pp. 150–162.
- [21] K. Zhou, M. W. Krentel, and J. Mellor-Crummey, "Tools for top-down performance analysis of gpu-accelerated applications," in *Proc. 34th ACM Int. Conf. Supercomputing*, New York, NY, USA: Association for Computing Machinery, 2020, doi: [10.1145/3392717.3392752](https://doi.org/10.1145/3392717.3392752).
- [22] G. Mao, D. Böhme, M.-A. Hermanns, M. Geimer, D. Lorenz, and F. Wolf, "Catching idlers with ease: A lightweight wait-state profiler for mpi programs," in *Proc. 21st Eur. MPI Users' Group Meeting*, New York, NY, USA: Association for Computing Machinery, 2014, pp. 103–108, doi: [10.1145/2642769.2642783](https://doi.org/10.1145/2642769.2642783).

- [23] O. H. Mondragon, P. G. Bridges, S. Levy, K. B. Ferreira, and P. Widener, "Understanding performance interference in next-generation hpc systems," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2016, pp. 384–395.
- [24] X. You, Z. Xuan, H. Yang, Z. Luan, Y. Liu, and D. Qian, "GVARP: Detecting performance variance on large-scale heterogeneous systems," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal. SC*, 2024, pp. 900–915.
- [25] K. B. Ferreira, P. Widener, S. Levy, D. Arnold, and T. Hoeffler, "Understanding the effects of communication and coordination on checkpointing at scale," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 883–894.
- [26] S. Levy, B. Topp, K. B. Ferreira, D. Arnold, T. Hoeffler, and P. Widener, "Using simulation to evaluate the performance of resilience strategies at scale," in *Proc. Int. Workshop Perform. Model. Benchmarking Simul. High Perform. Comput. Syst.*, Denver, CO, USA, 2014, pp. 91–114.
- [27] C. Yang, L. Zhou, H. Wen, Z. Zhou, and Y. Wu, "H-VGRAE: A hierarchical stochastic spatial-temporal embedding method for robust anomaly detection in dynamic networks," 2020, *arXiv: 2007.06903*.
- [28] F. Chen, "PUFFIN: An efficient DNN training accelerator for direct feedback alignment in fefet," in *Proc. IEEE/ACM Int. Symp. Low Power Electron. Des.*, 2021, pp. 1–6.
- [29] L. Cai et al., "Structural temporal graph neural networks for anomaly detection in dynamic graphs," in *Proc. 30th ACM Int. Conf. Inf. Knowl. Manage.*, New York, NY, USA: Association for Computing Machinery, 2021, pp. 3747–3756, doi: [10.1145/3459637.3481955](https://doi.org/10.1145/3459637.3481955).
- [30] D. Zhu, Y. Ma, and Y. Liu, "A flexible attentive temporal graph networks for anomaly detection in dynamic networks," in *Proc. IEEE 19th Int. Conf. Trust Secur Privacy Comput. Commun.*, 2020, pp. 870–875.
- [31] M. Chadha and M. Gerndt, "Modelling DVFS and UFS for region-based energy aware tuning of HPC applications," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2019, pp. 805–814.
- [32] K. O'brien, I. Pietri, R. Reddy, A. Lastovetsky, and R. Sakellariou, "A survey of power and energy predictive models in HPC systems and applications," *ACM Comput. Surv.*, vol. 50, no. 3, Jun. 2017, doi: [10.1145/3078811](https://doi.org/10.1145/3078811).
- [33] X. You, H. Yang, Z. Xuan, Z. Luan, and D. Qian, "Powerspector: Towards energy efficiency with calling-context-aware profiling," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2022, pp. 1272–1282.
- [34] P. Huang, B. Maag, T. Sivanthi, and C. Xing, "Work in progress: Early timing prediction of real-time tasks in continuous integration environments," in *Proc. IEEE 30th Real-Time Embedded Technol. Appl. Symp.*, 2024, pp. 390–393.
- [35] X. Zheng, P. Ravikumar, L. K. John, and A. Gerstlauer, "Learning-based analytical cross-platform performance prediction," in *Proc. Int. Conf. Embedded Comput. Systems: Architectures Model. Simul.*, 2015, pp. 52–59.
- [36] X. Zheng, L. K. John, and A. Gerstlauer, "Accurate phase-level cross-platform power and performance estimation," in *Proc. 53rd ACM/EDAC/IEEE Des. Automat. Conf.*, 2016, pp. 1–6.
- [37] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with PAPI-C," in *Proc. 3 rd Int. Workshop Parallel Tools High Perform. Comput.*, 2010, pp. 157–173.
- [38] T. Mikolov, "Efficient estimation of word representations in vector space," 2013, *arXiv:1301.3781*.
- [39] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016, *arXiv:1609.02907*.
- [40] O. Amine and M. Mestari, "Graph oriented attention networks," *IEEE Access*, vol. 12, pp. 47057–47067, 2024.
- [41] I. Sutskever, O. Vinyals, and Q. V. Le, *Sequence to Sequence Learning With Neural Networks*, Cambridge, MA, USA: MIT Press, 2014, pp. 3104–3112.
- [42] S. Bengio, O. Vinyals, N. Jaitly, and N. Shazeer, *Scheduled Sampling for Sequence Prediction With Recurrent Neural Networks*, Cambridge, MA, USA: MIT Press, 2015, pp. 1171–1179.
- [43] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," 2013. [Online]. Available: <https://api.semanticscholar.org/CorpusID>
- [44] D. H. Bailey, "NAS parallel benchmarks," in *Encyclopedia of Parallel Computing*, Berlin, Germany: Springer, pp. 1254–1259, 2011.
- [45] A. P. Thompson et al., "LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," *Comput. Phys. Commun.*, vol. 271, 2022, Art. no. 108171. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S001046521002836>
- [46] L. Adhianto et al., "HPCTOOLKIT: Tools for performance analysis of optimized parallel programs," *Concurrency Computation: Pract. Experience*, vol. 22, no. 6, pp. 685–701, 2010.
- [47] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The scalasca performance toolset architecture," *Concurrency Computation: Pract. Experience*, vol. 22, no. 6, pp. 702–719, 2010.
- [48] A. Knüpfer et al., "The vampir performance analysis tool-set," in *Proc. Proc. 2nd Int. Workshop Parallel Tools High Perform. Comput.*, Jul. 2008, HLRS, 2008, pp. 139–155.
- [49] L. Wei and J. Mellor-Crummey, "Using sample-based time series data for automated diagnosis of scalability losses in parallel programs," in *Proc. 25th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2020, pp. 144–159.
- [50] A. Knüpfer et al., "Score-P: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir," in *Proc. 5th Int. Workshop Parallel Tools High Perform. Comput.*, Dresden, 2012, pp. 79–91.
- [51] S. Shende, A. D. Malony, W. Spear, and K. Schuchardt, "Characterizing I/O performance using the tau performance system," in *Applications, Tools and Techniques on the Road to Exascale Computing*, Amsterdam, the Netherlands: IOS Press, 2012, pp. 647–655.
- [52] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. De Supinski, "Scalatrace: Scalable compression and replay of communication traces for high-performance computing," *J. Parallel Distrib. Comput.*, vol. 69, no. 8, pp. 696–710, 2009.
- [53] X. Wu and F. Mueller, "Elastic and scalable tracing and accurate replay of non-deterministic events," in *Proc. 27th Int. ACM Conf. Int. Conf. Supercomputing*, 2013, pp. 59–68.
- [54] C. Wang, P. Balaji, and M. Snir, "Pilgrim: Scalable and (near) lossless MPI tracing," in *Proc. Int. Conf. High Perform. Computing, Networking, Storage Anal.*, 2021, pp. 1–14.
- [55] J. Zhai, J. Hu, X. Tang, X. Ma, and W. Chen, "Cypress: Combining static and dynamic analysis for top-down communication trace compression," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 143–153.
- [56] R. Brightwell and K. D. Underwood, "An analysis of NIC resource usage for offloading MPI," in *Proc. 18th Int. Parallel Distrib. Process. Symp.*, 2004, Art. no. 183.
- [57] X. Wu and F. Mueller, "ScalaExtrap: Trace-based communication extrapolation for spmd programs," *ACM SIGPLAN Notices*, vol. 46, no. 8, pp. 113–122, 2011.
- [58] F. Petrini, D. J. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asc Q," in *Proc. ACM/IEEE Conf. Supercomputing*, 2003, Art. no. 55.
- [59] M. Hidayetoğlu et al., "MemXCT: Memory-centric X-ray ct reconstruction with massive parallelization," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2019, pp. 1–56.
- [60] H. S. Gunawi et al., "Fail-slow at scale: Evidence of hardware performance faults in large production systems," *ACM Trans. Storage*, vol. 14, no. 3, pp. 1–26, 2018.
- [61] O. H. Mondragon, P. G. Bridges, S. Levy, K. B. Ferreira, and P. Widener, "Scheduling in-situ analytics in next-generation applications," in *Proc. 16th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2016, pp. 102–105.
- [62] F. Zheng et al., "Goldrush: Resource efficient in situ scientific data analytics using fine-grained interference aware execution," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2013, pp. 1–12.
- [63] K. Pedretti, S. L. Olivier, K. B. Ferreira, G. Shipman, and W. Shu, "Early experiences with node-level power capping on the Cray XC40 platform," in *Proc. 3 rd Int. Workshop Energy Efficient Supercomputing*, 2015, pp. 1–10.



Zhibo Xuan is currently working toward the PhD degree with the School of Computer Science and Engineering, Beihang University. He is currently working on performance analysis, machine learning-based performance analysis, and performance optimization. His research interests include performance analysis, HPC, and deep learning.



Xin Sun received the BS degree from the School of Computer Science and Information Engineering, Hefei University of Technology, in 2022. She currently working toward the MS degree with the School of Computer Science and Engineering, Beihang University. Her research interests include performance analysis and high-performance computing (HPC).



Zhongzhi Luan (Senior Member, IEEE) received the PhD degree from the School of Computer Science, Xi'an Jiaotong University. He is an associate professor of Computer Science and Engineering, and assistant director with the Sino-German Joint Software Institute (JSI) Laboratory at Beihang University, China. Since 2003, His research interests including distributed computing, parallel computing, grid computing, HPC, and the new generation of network technology.



Xin You received the PhD degree from the School of Computer Science and Engineering, Beihang University, in 2024. He is currently working toward the graduate degree with the School of Computer Science and Engineering, Beihang University. He is currently working on performance analysis and performance optimization. His research interests include performance analysis, HPC, and deep learning.



Yi Liu received the PhD degree from the Department of Computer Science, Xi'an Jiaotong University, China. He is a professor with the School of Computer Science and Engineering, and director of the Sino-German Joint Software Institute (JSI), Beihang University, China. His research interests include computer architecture, HPC, and new generation of network technology.



Hailong Yang (Member, IEEE) received the PhD degree from the School of Computer Science and Engineering, Beihang University, in 2014. He is a professor with the School of Computer Science and Engineering, Beihang University. He has been involved in several scientific projects such as performance analysis for big data systems and performance optimization for large scale applications. His research interests include parallel and distributed computing, HPC, performance optimization, and energy efficiency.



Depei Qian received the master's degree from the University of North Texas, in 1984. He is a professor with the Department of Computer Science and Engineering, Beihang University, China. He is currently serving as the chief scientist with China National High Technology Program (863 Program) on high productivity computer and service environment. He is also a fellow of China Computer Federation (CCF). His research interests include innovative technologies in distributed computing, high performance computing, and computer architecture.