SMART: A Surrogate Model for Predicting Application Runtime in Dragonfly Systems

Xin Wang[*,1] **Pietro Lodi Rizzini**[*,1] **Sourav Medya,**[1] **Zhiling Lan**[1,2]

[1]University of Illinois Chicago
[2]Argonne National Laboratory
{xwang823, plodir2, medya, zlan}@uic.edu

Abstract

The Dragonfly network, with its high-radix and low-diameter structure, is a leading interconnect in high-performance computing. A major challenge is workload interference on shared network links. Parallel discrete event simulation (PDES) is commonly used to analyze workload interference. However, high-fidelity PDES is computationally expensive, making it impractical for large-scale or real-time scenarios. Hybrid simulation that incorporates data-driven surrogate models offers a promising alternative, especially for forecasting application runtime, a task complicated by the dynamic behavior of network traffic. We present SMART, a surrogate model that combines graph neural networks (GNNs) and large language models (LLMs) to capture both spatial and temporal patterns from port level router data. SMART outperforms existing statistical and machine learning baselines, enabling accurate runtime prediction and supporting efficient hybrid simulation of Dragonfly networks.

Code — https://github.com/SPEAR-UIC/SMART
Datasets — https://zenodo.org/records/16667461

Introduction

High-performance computing (HPC) systems play a vital role in numerous scientific domains, including climate modeling and drug discovery (Zhu and et al. 2014), where complex simulations and data-intensive computations are required. In these systems, the interconnect network serves as the central nervous system, facilitating data exchange among computer nodes and hence determining the system's overall performance. Among the various network topologies designed to meet these demands, the Dragonfly topology stands out as a high-radix, low-diameter solution that balances cost and performance (Kim and et al. 2008). Distinct from fat-tree networks, Dragonfly networks reduce infrastructure costs while maintaining high throughput and scalability. This effectiveness is reflected in their widespread adoption in the fast supercomputers, as evidenced by the latest Top500 list (6/2025), where six of the top ten supercomputers utilize Dragonfly interconnects (top500.org 2024).

In Dragonfly systems, a critical challenge is workload interference, which arises as multiple applications run concurrently on the same machine, producing highly dynamic and complex network traffic patterns. Parallel discrete event simulation (PDES) is widely used to model workload interference and network flows with high fidelity, i.e., at the flit-level granularity (Fujimoto 1990). CODES is a widely recognized PDES-based network simulator known for its ability to accurately model Dragonfly network behavior in detail(Mubarak et al. 2017; Carothers and et al. 2002; Wang et al. 2020). However, the computational complexity of high-fidelity PDES grows intractably, requiring the processing of billions of flit-level events across thousands of routers. As a result, hybrid simulation integrating data-driven surrogate models into PDES has become increasingly important for accelerating network modeling while preserving accuracy. One particular area of interest is constructing surrogate models to predict application runtime in Dragonfly systems.

Developing *data-driven surrogate models* for PDES is extremely challenging due to the highly dynamic nature of network traffic (e.g., at the millisecond scale) and the complex performance fluctuations inherent in large-scale systems. These challenges are exacerbated when multiple parallel applications run concurrently, competing for shared network resources, with iteration times that shift dynamically in response to changing network loads. Additionally, surrogate models must strike a delicate balance: they must achieve high accuracy while imposing minimal runtime overhead to be practical for real-time use. Such surrogate models hold significant potential to enhance networking systems by enabling improved decision-making in areas such as routing, congestion management, and resource allocation.

In this work, we explore advanced machine learning for surrogate modeling of Dragonfly interconnects, aiming to advance PDES and, in turn, contribute meaningful advancements to Dragonfly network research. The major contributions are summarized as follows.

- **Objective:** Our goal is to develop an accurate yet lightweight surrogate model for predicting application iteration times in Dragonfly. Such a model can not only accelerate networking simulations, but also contributes to improving networking decisions in Dragonfly systems.
- **Novel Approach:** We develop **SMART** (**S**urrogate **M**odel for Predicting **A**pplication **R**un**T**ime). Our design

[*]These authors contributed equally.

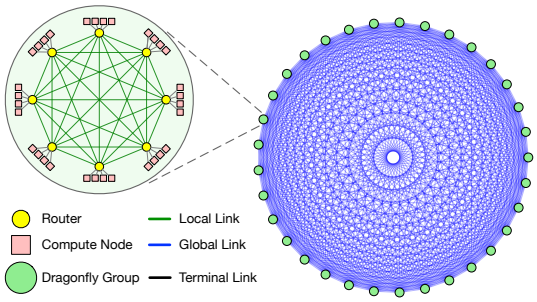


Figure 1: Topology of a 1,056-node 1D Dragonfly network.

leverages domain-specific knowledge of Dragonfly interconnects by incorporating both *topological structure* and *temporal dynamics* of network traffic to address the application runtime forecasting problem. Unlike the conventional algorithms, SMART integrates graph neural networks (GNNs) and large language models (LLMs) to holistically model the complex behavior of Dragonfly systems. Specifically, GNNs are employed to capture the hierarchical and spatial connectivity inherent in the Dragonfly topology, while LLMs are introduced to enhance temporal modeling through long-range pattern recognition and contextual prompting, a capability unavailable in traditional sequence models. To the best of our knowledge, this is the first effort of building a surrogate model for large-scale networking using LLMs and GNNs.

- **Experimental Results:** We conduct extensive experiments using two datasets generated from a 1,056-node Dragonfly system, where multiple representative HPC workloads are executed concurrently under various job placement strategies. Our method SMART significantly outperforms all the baselines in predictive accuracy. Moreover, SMART achieves an inference time of just 0.515 seconds, offering at least an order-of-magnitude speedup over the original simulation time.

Background

Dragonfly Network Topology. The Dragonfly topology is widely adopted in exascale HPC systems, including those in the TOP500 list (top500.org 2024). It offers high bandwidth and low diameter while balancing scalability and cost. As shown in Figure 1, Dragonfly (Kim and et al. 2008) is organized into groups of routers connected via local links, with compute nodes attached through terminal ports. Groups are interconnected in an all-to-all pattern using global links, enabling low-hop communication across the system. We focus on the 1D Dragonfly variant, where routers within each group are fully connected. This design underlies Slingshot networks used in production systems such as Frontier and Aurora (ORNL 2022; Stevens and et al. 2019), making it a relevant target for our study.

Hyrbid Network Simulation. Parallel discrete event simulation (PDES) is widely used for high fidelity modeling of large scale network systems. CODES, built on the ROSS engine, enables flit level simulation and has been applied to various HPC interconnect studies (Mubarak et al. 2017; Yang et al. 2016a,b; Wolfe et al. 2017).

Due to the high computational cost of PDES, hybrid approaches combine it with machine learning surrogate models to improve scalability (Cruz-Camacho et al. 2024). Figure 2, shows this process with four stages: (1) simulation data is collected, (2) used to train or fine tune the surrogate model, (3) the surrogate is validated through real time PDES output, and (4) a control loop decides whether to continue with PDES or switch to the faster surrogate. This adaptive feedback mechanism enables accurate yet efficient simulation.

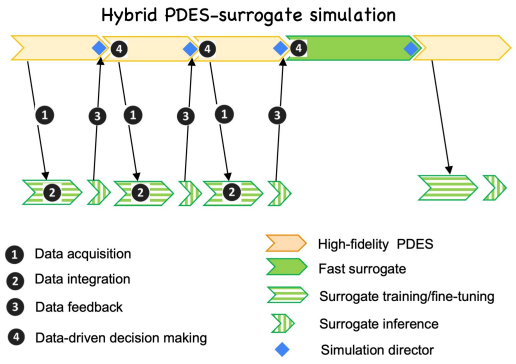


Figure 2: Hybrid PDES-surrogate simulation (Rizzini 2024).

Related Work

Application runtime forecasting. Time series prediction has been widely studied in statistics and machine learning, with techniques such as ARIMA (Autoregressive Integrated Moving Average) (A. and C. 1979), Multi-Layer Perceptrons (Shiblee and et al. 2009), Convolutional Neural Networks (Wibawa and et al. 2022), RNNs (Recurrent Neural Networks) (Rumelhart and et al. 1987), and LSTMs (Long Short-Term Memory) (Hochreiter and et al."urgen 1997). These methods primarily focus on temporal data, often neglecting the spatial dependencies inherent in systems like HPC networks. Moreover, graph neural networks (GNNs) have been used to estimate metrics such as delay, jitter, and packet loss, as well as to enhance modeling expressiveness and granularity (Suárez-Varela et al. 2022; Ferriol-Galmés et al. 2023; Wang et al. 2022). A recent effort (Rizzini 2024) applied the Diffusion Convolutional Recurrent Neural Network (DCRNN) to predict application iteration times on a 72-node Dragonfly system. Different from this, we propose a novel integration of graph neural networks and large language models (LLMs) for more accurate predictions.

Large Language Models. Large Language Models (LLMs) have recently gained significant attention for their ability to handle a wide range of tasks, including the ones on time-series data. More specifically, LLMs have been explored for their potential to capture complex temporal patterns without requiring fine-tuning. Recent studies (Das et al. 2024; Jin et al. 2024; Chen et al. 2024; Garza, Challu, and Mergenthaler-Canseco 2024) have demonstrated the effectiveness of LLMs in time series prediction by leveraging their inherent ability to model sequential data and their capacity to generalize across domains. In contrast, our ap-

proach integrates LLMs and GNNs to capture both temporal and spatial dependencies in HPC systems simulations.

Graph Neural Networks (GNNs). GNNs have emerged as a powerful class of models for learning over graph-structured data. Differently from traditional deep learning models that work on data in the Euclidean domain, GNNs can handle irregular data that can be represented with a graph. Examples are social networks, molecular graphs, and communication networks. GNNs operate by aggregating information from a node's neighbors: this enables them to capture both local and global structural dependencies and makes them effective on relational data. One of the approaches is the GraphSAGE (Hamilton, Ying, and Leskovec 2018), which generalizes GNNs to large-scale graphs by sampling and aggregating features from a node's local neighborhood. Another popular model is the Graph Attention Network (GAT) (Veličković et al. 2018), which introduces attention mechanisms to weigh the importance of neighboring nodes dynamically. Recent advances in GNNs have led to the development of models that can handle graph structures that evolve over time. These models usually combine GNNs with sequential models, such as RNNs (Rumelhart and et al. 1987) or Transformers (Vaswani et al. 2023). For example, the Temporal Graph Network (TGN) (Rossi et al. 2020) extends GNNs to handle continuous-time dynamic graphs, making it suitable for applications like social network analysis and recommendation systems.

Graph-based Methods for Application Runtime Forecasting. Graph-based learning has gained traction in network performance modeling, enabling fine-grained representations of network structure and dynamics. Previous studies have used GNNs to estimate flow level metrics (Suárez-Varela et al. 2022; Ferriol-Galmés et al. 2023; Wang et al. 2022), often in the context of generic communication or datacenter networks. Our focus is on Dragonfly, an interconnect network commonly used in HPC systems, that employs adaptive routing that dynamically adjusts paths based on network conditions. Because of this, network interference and application variability are significantly more complex and larger in Dragonfly networks, which motivates the need for specialized modeling in this study. A step toward this goal was taken by applying DCRNN to runtime prediction (Rizzini 2024), but the effectiveness of the model diminishes when scaled beyond small networks. Moreover, recent works such as PraNet (Liu et al. 2024) model short-term traffic bursts in generic internet or metaverse simulations using GNNs, transformers, and queuing theory with limited simulation data. In contrast, SMART focuses on high-fidelity simulation traces from HPC interconnects and proposes the first integration of GNNs and LLMs for spatio-temporal modeling of application runtime. This combination enables the model to generalize across job placement strategies, workloads, and dynamic traffic patterns, offering practical accuracy and inference efficiency at scale.

SMART advances this direction by combining GNNs with LLMs to jointly model spatial structure and temporal dynamics from port-level simulation data. To our knowledge, this is the first surrogate model integrating GNNs and LLMs for large-scale HPC runtime forecasting.

Our Methodology

Networking Data

High-fidelity simulations generate two main datasets: (1) *application data*, which captures workload characteristics, and (2) *router data*, which records port-level features every $250\,\mu s$. These are used to construct a *temporal graph* representation.

The temporal graph is represented as a sequence $G_1, G_2, \ldots, G_T$, where each graph G_i corresponds to iteration i. The node set V and edge set E remain fixed, while node features change over time. In the graphs, *nodes* represent the router ports of the system, with some ports also representing the computing devices directly connected to them in the Dragonfly topology. *Edges* are defined in two ways: (1) fully connecting nodes accounting for ports of the same router and (2) connecting ports connected by global or local links. This results in a complete graph structure that captures the connectivity of a dragonfly system.

Node features. Each node is annotated with features that describe the network state at the corresponding port during each iteration. Since network snapshots are taken every $250\,\mu s$, we aggregate the snapshots over the iteration period by statistics like minimum, maximum, average or a quantile.

Active nodes are Dragonfly computing units executing workloads whose iteration times are being predicted. For these nodes, aggregation bounds are defined by the start and end timestamps of the iteration. For non-active nodes, the aggregation interval $[lb, ub)$ is determined by: (1) if one of the terminal ports is active, the iteration timestamps of that node define the bounds, (2) if all terminal ports are active, the bounds are set by the earliest start and the latest end timestamps across the iterations, (3) if none of the terminal ports is active, the bounds are derived from the nearest active node's timestamps.

Problem Definition

Let the Dragonfly network consist of n_r routers, N_c computer nodes, and application has N_p processes distributed across the computer nodes for execution. During execution, the N_p processes collaborate to complete T iterations. For each process $p \in \{1, 2, \ldots, N_p\}$ at a specific iteration $t \in \{1, 2, \ldots, T\}$, we define $y_{p,t}$ as the time taken for its application iteration and $x_{p,t}$ as the network characteristics of the router connected to the compute node hosting the rank p. To simplify the notation, we drop the subscript p in the following descriptions. We formally define the problem below.

Problem Statement. Given a sequence of application iteration times y and network characteristics x for process rank p within a look-back window, we define $\mathcal{B}$ as the input:

$$\mathcal{B} = \{y_{t-(L_y-1)}, y_{t-(L_y-2)}, \ldots, y_t; \\ x_{t-(L_x-1)}, x_{t-(L_x-2)}, \ldots, x_t\}$$

consisting of look-back windows of length L_y for y and L_x for x. The goal is to predict the future application iteration times at $t+1$, denoted as y_{t+1}.

Table 1: Notations used in the model description

Symbol	Description
G_t	Graph representing the system at iteration t
V	Set of all nodes (router ports) in the graph
V_a	Subset of V corresponding to active nodes
E	Set of edges (connections between router ports)
$X^{(t)}$	Node feature matrix at iteration t: $(\|V\| \times d_f)$
y_t	Application iteration times for iteration t
T_{inGNN}	Length of the look-back window for GNN input
T_{inLLM}	Length of the look-back window for LLM input
$H^{(t)}$	Node embedding matrix from the GNN encoder: $(\|V\| \times d_h)$
$Z^{(t)}$	Temporal node embeddings from the Transformer:$(\|V\| \times d_z)$
$E^{(t)}$	LLM-generated embeddings for active nodes: $(\|V_a\| \times d_{llm})$
$F^{(t)}$	Reduced dimensionality LLM embeddings $(\|V_a\| \times d_z)$
d_f	Number of features per node in $X^{(t)}$
d_h	Dimensionality of GNN embeddings
d_z	Dimensionality of temporal embeddings from Transformer
d_{llm}	Dimensionality of LLM hidden state

Our Model: SMART

Our model, SMART, has three components, GNN, Transformer, and LLM, integrated for spatio-temporal modeling of workload contention and temporal variation in Dragonfly systems, which conventional graph or time series models cannot capture. Figure 3 shows the architecture. Table 1 summarizes the notations.

GNN Encoder Our first goal is to capture the topological aspect of the dragonfly network with a graph neural network (GNN)-based encoder. The GNN encoder takes a sequence of temporal graphs $\{G_{t-T_{inGNN}}, G_{t-T_{inGNN}+1}, \ldots, G_{t-1}\}$ within a look-back window of length T_{inGNN} where each graph $G_t = (V, E, X^{(t)})$ represents the network state at iteration t. Here, V is the set of nodes (router ports), E is the set of edges (connections between ports), and $X^{(t)} \in \mathbb{R}^{|V| \times d_f}$ is the node feature matrix at iteration t, with d_f being the number of features per node. The GNN Encoder is a Graph Convolutional Network (GCN) (Kipf and Welling 2017) which is applied to each graph G_t to capture individual spatial dependencies. The propagation rule for layer $l+1$ in GCN is given by:

$$H^{(l+1,t)} = \sigma\left(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}H^{(l,t)}W^{(l)}\right)$$

where l is the layer, t is the iteration number, $\tilde{A} = A + I_N$ is the adjacency matrix A with self-loops, $\tilde{D}$ is the diagonal degree matrix of $\tilde{A}$, $H^{(l)}$ represents the node feature matrix at layer l, with $H^{(0,t)} = X^{(t)}$, $W^{(l)}$ is the trainable weight matrix, $\sigma(\cdot)$ denotes the RELU activation function. The output of the GNN Encoder is a sequence of node embeddings that capture the spatial dependencies in the network at each input iteration: $\{H^{(1)}, H^{(2)}, \ldots, H^{(T_{inGNN})}\}$, where $H^{(t)} \in \mathbb{R}^{|V| \times d_h}$ is the node embedding matrix at iteration t, and d_h is the dimensionality of the embeddings.

Temporal transformer to encode temporal dependencies Our next goal is to capture the temporal dependencies in a unified way. SMART utilizes a transformer (Vaswani et al. 2023) module to process the temporal graph embeddings

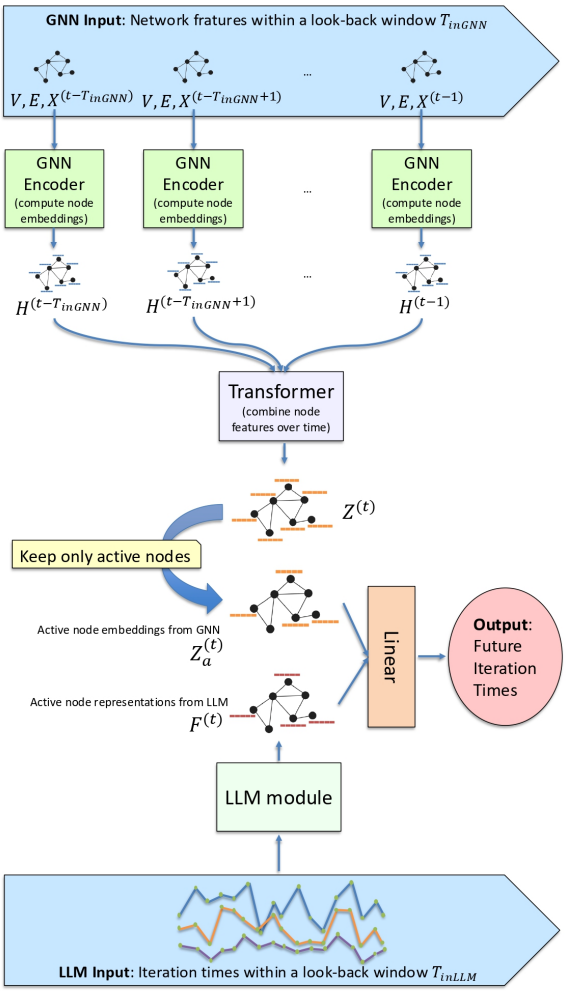


Figure 3: The architecture of our proposed model, SMART. The GNN Encoder (top) generates the node embeddings while the Transformer helps to produce node representations over time. The lower part shows the LLM-based component. The outputs of each component is concatenated node-wise and fed into a Linear layer for the final prediction.

$\{H^{(1)}, H^{(2)}, \ldots, H^{(T_{inGNN})}\}$ generated by the GNN encoder. Unlike traditional recurrent or convolutional neural networks, transformers are designed for sequence handling via self-attention. Here we employ transformer to capture the temporal dependencies in the graph embeddings from each iteration. The input to the transformer encoder is a sequence of node embeddings over time, enriched with positional encoding. The transformer module processes this input and applies an attention mechanism.

Time-dependent Attention. The core operation of the transformer module in SMART is the attention mechanism, which computes the importance of different parts of the input sequence. The input embedding of size (T_{inGNN}, d_m), where T_{inGNN} is the length of the input sequence and $d_m (= d_h \times |V|)$ is its dimensionality, is linearly transformed into three matrices: Query (Q), Key (K), and Value (V).

These matrices are computed as:

$$Q = HW^Q, \quad K = HW^K, \quad V = HW^V$$

where H in our setting is as follows:

$$H = [H^{(t-T_{inGNN})}, H^{(t-T_{inGNN}+1)}, \ldots, H^{(t-1)}]$$

and $H \in \mathbb{R}^{T_{inGNN} \times |V| \times d_h}$. Moreover, W^Q, W^K, W^V are trainable weight matrices. Each attention head calculates scaled dot-product attention as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

where d_k is the dimensionality of K. Multi-head attention is concatenation of the outputs of multiple attention heads:

$$\text{multihead}(Q, K, V) = \text{concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

Here, $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$, and W_i^Q, W_i^K, W_i^V, and W_i^O are learnable weight matrices. This mechanism allows the transformer to focus on different parts of the sequence simultaneously to capture diverse dependencies. The output of the temporal transformer is a sequence of node embeddings that encode both spatial—by the GNN encoder—and temporal dependencies within the input features look-back window.

Output of the Transformer Module. The final output from the encoder is passed to the decoder, together with the last temporal node embeddings. These embeddings represent the most recent state of the nodes and provide critical information for predicting the next temporal embedding. In the decoder, the target sequence (last temporal node embeddings) is shifted to the right, ensuring that the model does not access the embedding it is trying to predict. The shifted target and the output of the encoder are then fed into the second and third layers, which correspond to successive encoder blocks, where K and V are derived from the encoder outputs, and Q is derived from the target. The output is finally passed to a dense layer in order to produce the temporal node embeddings. The final output from the transformer is $Z^{(t)} \in \mathbb{R}^{|V| \times d_z}$.

LLM-powered Forecasting We utilize a large language model (LLM)-based approach to further enhance our model, SMART. Unlike traditional sequence models, LLMs in SMART enable contextual prompting, incorporating task-specific and domain knowledge alongside time-series data. This helps capture long-range dependencies and improves runtime prediction across diverse workloads. To the best of our knowledge this is the first work that uses an LLM in the surrogate model to forecast application runtime. In SMART, this component uses the historical application iteration time data in as a sequence $\{y_{t-T_{inLLM}}, y_{t-T_{inLLM}+1}, \ldots, y_t\}$ within a pre-defined look-back window T_{inLLM}.

We closely follow the mechanism of Time-LLM (Jin et al. 2024). The steps of this module is as follows. The input sequence is first patched to normalize the input time series. Next we embed them via a *patch embedding* module into a dimension that is compatible with the LLM via a simple linear layer. The patches are then transformed via a multi-head attention layer to obtain the embeddings which are in the same space as the pre-trained LLM. Finally, the LLMs generate embeddings $E^{(t)} \in \mathbb{R}^{|V_a| \times d_{llm}}$ to represent the temporal patterns in the iteration times. Here, V_a represents the subset of V corresponding to the active nodes and d_{llm} is the number of hidden channels in the underlying LLM model.

Prompting. We use *Prompt-as-Prefix (PaP)* technique for prompting (Jin et al. 2024). In this technique, the patch embeddings are prefixed with prompts that contain context to the input, including task instructions, domain knowledge, and descriptive statistics about the time series. For example, statistics such as minimum, maximum, median values, and trends can be used for this purpose. We use the following prompt template given below, where $workload\_name$ is the name of the application workload, T_{inLLM} is the length of the historical input window to the LLM component.

Prompt for Forecasting

The dataset contains application iteration times for the $[workload\_name]$ workload.
Task description: Forecast the next step given the previous $[T_{inLLM}]$ steps information.
Input statistics: min value [...], max value [...], median value [...]

The input—which is composed of the prefix prompt and embedded patches—fed into the LLM. The obtained LLM output $E^{(t)} \in \mathbb{R}^{|V_a| \times d_{llm}}$ is flattened and linearly projected into an hidden dimension that matches the temporal node embeddings size from the transformer module, resulting in a final output $F^{(t)} \in \mathbb{R}^{|V_a| \times d_z}$.

Final Integration In the final step, we integrate GNN embeddings along with the LLM output effectively. The temporal embeddings $Z^{(t)} \in \mathbb{R}^{|V| \times d_z}$ from the Transformer are filtered by applying a node mask to ensure that only active nodes—such as those with target values—are considered in the merging process. The result of this operation is $Z_a^{(t)} \in \mathbb{R}^{|V_a| \times d_z}$. For each active node, the reduced-dimensional hidden state from the LLM is concatenated with the corresponding node embedding from the GNN. This creates a combined feature vector that encodes both spatial and temporal information powered by the LLM. The combined feature vector is passed through a fully connected layer to produce the final predictions ($\hat{y}_{t+1}$) for each node.Rather than manually tuning the balance between spatial and temporal modeling, SMART learns this integration end-to-end. The final prediction layer combines GNN-derived spatial features with LLM-derived temporal context, allowing the model to automatically determine their relative importance.

Online Tuning Strategy We develop an online tuning strategy to address the evolving nature of network traffic and workload behavior in Dragonfly systems. This approach is designed for hybrid simulation settings, where ground-truth data from PDES is intermittently available. The model

is fine-tuned during inference only when such feedback is present, rather than assumed to be continuous. With a feedback loop, SMART adjusts its parameters to align with the latest network dynamics, thus maintaining prediction accuracy over time. This strategy is particularly beneficial in environments where workloads are heterogeneous and exhibit temporal variability. Our approach has two phases. The first one is the offline learning over a training dataset consisting of 30% of the available data. The second phase is the inference phase. Here we consider the ground truth data to be available for every F_t iterations. This implies that if the iteration index t is multiple of F_t, we use the data from $t-F_t$ to t to update the model weights via the usual back-propagation. In the following, $F_t = \infty$ means that the online tuning strategy is turned off, i.e. the inference phase does not incorporate any update of the model weights.

Experiments

Table 2: Model hyperparameters

Parameter	Value
Patch length in LLM	2
Stride in LLM	1
LLM hidden channels D_{llm}	768
LLM num hidden layers	32
LLM model	GPT-2
GCN layers	2
GCN embedding size d_h	128
Transformer model dimension d_z	128
Number of attention heads in Transformer	8
Transformer encoder layers	2
Transformer decoder layers	2

We demonstrate the effectiveness of SMART in predicting application runtime on a 1,056 node Dragonfly system (Figure 1), where multiple applications execute concurrently and contend for shared network resources. Our evaluation examines prediction quality across a range of workloads and placement policies, and measures the inference time overhead to confirm that SMART is suitable for integration into hybrid simulation workflows. We further conduct ablation studies to quantify the contributions of the GNN and Time LLM components, analyze model sensitivity through hyperparameter variation, and compare SMART against several state of the art temporal forecasting models. The code and datasets used in this study are available as noted after the abstract.

Experimental Setup

Data Generation. We analyze two datasets from CODES simulations that capture the execution of multiple HPC applications running concurrently on a 1,056-node Dragonfly system (CODES 2025). We select several representative HPC applications spanning computational physics, molecular dynamics, and structured grid-based computations.

Dataset D_1: The first dataset (D_1) includes two application workloads: MILC (mil 2024) and LAMMPS (lam 2024). In this dataset, 512 nodes run MILC, another 512 nodes run LAMMPS, and the remaining 36 nodes generate Uniform Random (UR) background traffic, where each node sends successive messages to random destinations. MILC is an HPC application used for quantum chromodynamics (QCD) simulations, while LAMMPS is a molecular dynamics code focused on materials modeling.

Dataset D_2: The second dataset (D_2) extends D_1 by adding a third workload, NN, which represents a 27-point 3D stencil computation used for modeling iterative updates over a structured grid. Stencil computations are a fundamental pattern in HPC workloads, commonly used in scientific simulations, computational fluid dynamics, and iterative solvers. In this setup, 384 nodes run MILC, 512 nodes run LAMMPS, and 160 nodes run NN.

Job placement policies. Since application performance is significantly affected by different job placement policies(Yang et al. 2016a), we consider two widely used strategies for each dataset: *contiguous* and *random* job placement. In the random placement strategy, workload-assigned compute nodes are distributed across the system, whereas in the contiguous placement strategy, they are allocated sequentially. Thus, we evaluate our model using two datasets, each with two different job placement policies.

Consequently, our experiments encompass four different application and job placement combinations: **D1-Rand** (Dataset D1 under random placement), **D1-Cont** (Dataset D1 under contiguous placement), **D2-Rand** (Dataset D2 under random placement), and **D2-Cont** (Dataset D2 under contiguous placement).

The D_1 simulations were conducted on a Bebop Broadwell node (LCRC 2024), equipped with 36 Intel Xeon E5-2695v4 CPUs and 128GB of memory, with execution times of 66.17 hours for random placement and 38.68 hours for contiguous placement. The D_2 simulations were run on a compute_icelake_r650 machine on Chameleon(Keahey et al. 2020), featuring two Intel(R) Xeon(R) Platinum 8380 CPUs @ 2.30GHz and 256 GiB of memory, with execution times of 62.46 hours for random placement and 36.37 hours for contiguous placement.

Baselines. We use the following baselines:

- **Last** is a heuristic that forecasts the next iteration time as the last historical available one, i.e. $\hat{y}_t = \hat{y}_{t+1} = \hat{y}_{t+2} = ... = \hat{y}_{t+T_{out}} = y_{t-1}$.
- **Mean** forecasts based on the past W average iteration times, i.e. $\hat{y}_t = \hat{y}_{t+1} = \hat{y}_{t+2} = ... = \hat{y}_{t+T_{out}} = \frac{\sum_{i=t-W}^{t-1} y_i}{W}$. In the experiments, we use $W = T_{inGNN}$.
- **LSTM** (Long-Short Term Memory (Hochreiter and et al."urgen 1997)) predicts the next iteration time based on the historical iteration time values within a look-back window varied according to T_{inGNN}.
- **DCRNN-based Baseline.** This recent method (Rizzini 2024) uses a popular GNN-based architecture named as Diffusion Convolutional Recurrent Neural Network (DCRNN). To have a fair comparison, the length of the look-back window is varied according to T_{inGNN}.

Table 3: MAPE comparison of SMART versus other baselines for different historical input windows T_{inGNN}, T_{inLLM} and retrain frequencies F_T over the simulation in D_1. Bold indicates the best result, underlined indicates the second best (column-wise). SMART consistently outperforms the baselines.

SMART			T_{inGNN}											
			2				4				8			
			Cont		Rand		Cont		Rand		Cont		Rand	
		Tuning frequency F_t	MILC	LAMMPS	MILC	LAMMPS	MILC	LAMMPS	MILC	LAMMPS	MILC	LAMMPS	MILC	LAMMPS
T_{inLLM}	2	∞	3.98	2.64	3.9	2.54	4.30	2.64	4.17	2.59	4.64	2.88	4.30	2.67
		8	3.70	2.51	3.81	2.47	3.91	2.62	4.04	2.53	4.36	2.80	4.12	2.60
	4	∞	3.87	1.90	3.72	1.81	4.22	1.94	3.91	1.85	4.28	2.04	3.96	1.90
		8	3.75	1.86	3.38	1.78	4.05	**1.92**	3.52	**1.82**	4.18	**1.97**	3.59	**1.88**
	8	∞	3.50	1.89	3.40	1.87	4.04	2.06	3.67	1.89	3.90	2.05	3.71	1.95
		8	**3.19**	**1.78**	**3.12**	**1.84**	**3.76**	1.96	**3.34**	1.86	**3.74**	2.03	**3.44**	1.91
Baselines														
LSTM		∞	6.15	7.29	6.11	7.32	5.90	7.20	5.89	7.16	5.90	7.20	5.88	6.54
		8	6.09	7.25	6.05	7.28	5.82	7.15	5.81	7.11	5.79	6.48	5.83	6.53
DCRNN		∞	6.27	7.14	6.04	7.13	6.07	7.36	6.07	7.35	6.11	6.53	5.99	6.86
		8	6.35	7.10	5.98	7.10	5.97	7.34	5.97	7.33	6.00	6.49	5.88	6.82
LAST			7.86	8.14	7.83	8.21	7.86	8.14	7.83	8.21	7.86	8.14	7.83	8.21
MEAN			7.80	8.24	9.31	7.72	10.37	12.85	10.96	13.14	12.86	13.11	13.15	13.53

Performance Mesaures. To evaluate the performance of the models, we consider two key measures: (1) forecasting quality and (2) inference time. We use the standard Mean Absolute Percentage Error (MAPE) as the metric to assess quality, which is computed as: MAPE $= \frac{1}{N}\sum_{i=1}^{N}\left|\frac{y_i-\hat{y}_i}{y_i}\right| * 100$ where y_i represents the actual iteration time, and $\hat{y}_i$ represents the predicted iteration time.

Other Settings. For each application, dataset and job placement strategy, we train and test the models tailored to predict the specific application iteration times. Table 2 shows all the hyperparameter values that are used in our experiments. We run the experiments for different combinations of $T_{inLLM} = 2, 4, 8$ and $T_{inGNN} = 2, 4, 8$ and tuning frequencies $F_T = \infty, 8$, where $F_T = \infty$ means that the model is never tuned during the testing, while $F_T = 8$ means that every 8 test iterations, the model weights are updated to allow the model to capture recent changes in the network patterns. Training each SMART model takes approximately 3–4 hours using two NVIDIA A100-PCIE GPUs (40 GiB memory each) with CUDA 12.4. Training is a one-time cost, while inference is lightweight and efficient, as detailed in Table 2 of the main paper. All reported results are based on a single run per experimental setting.

Results on Forecasting Quality

This section evaluates the forecasting quality of SMART alongside baseline models for predicting iteration time. Table 3 and Table 4 present the MAPE values for experiments conducted on datasets D_1 and D_2, respectively. The columns "MILC", "LAMMPS" and "NN" refer to models trained considering the nodes corresponding to the specific application as active. The columns "Cont" and "Rand" indicate whether the dataset follows a contiguous or random placement strategy. The cells between "T_{inLLM}" and "T_{inGNN}" refer to the results for the proposed SMART model, the other cells refer to the results for the baselines.

The results show that the best performance is obtained for $T_{inLLM} = 8, T_{inGNN} = 2$ in all the cases. This suggests the LLM effectively captures long-term temporal patterns while the GNN focuses on recent spatial dynamics. In addition, the online tuning strategy presented for the DCRNN model in (Rizzini 2024) appears to be beneficial. Incorporating an online tuning strategy ($F_t = 8$) consistently reduces forecasting errors compared to a static model ($F_t = \infty$). By updating the model weights every 8 iterations, SMART adapts to recent changes in network behavior, resulting in lower MAPE values in both data sets and placement types.

Table 5 summarizes key statistics on application iteration times and prediction errors for MILC, LAMMPS, and NN in the D_2 dataset under both contiguous and random job placement strategies. The table reports the maximum and minimum iteration times, the standard deviation, as well as the Mean Absolute Percentage Error (MAPE) and Root Mean Square Error (RMSE) of the SMART predictions. LAMMPS exhibits a significantly larger variance in iteration time compared to MILC and NN. However, SMART effectively adapts to these fluctuations. These detailed statistics further validate SMART's robustness in handling execution time variability caused by network interference. Additionally, the experimental results show minimal accuracy differences between contiguous and random placements: MILC exhibits a MAPE difference of 0.12%, and LAMMPS shows an even narrower gap of 0.02%. However, random placement intro-

Table 4: MAPE comparison for different historical input windows T_{inGNN}, T_{inLLM} and retrain frequencies F_T over the simulation in D_2. Bold indicates the best result, underlined indicates the second best (column-wise). SMART consistently outperforms the baselines; the best result is often obtained with longer LLM input window ($T_{inLLM} = 8$).

SMART			T_{inGNN}																	
			2						4						8					
			Cont				Rand		Cont				Rand		Cont				Rand	
T_{inLLM}		Tuning frequency F_t	MILC	LAMMPS	NN	MILC	LAMMPS	NN	MILC	LAMMPS	NN	MILC	LAMMPS	NN	MILC	LAMMPS	NN	MILC	LAMMPS	NN
2		∞	4.29	2.81	3.30	4.04	2.63	3.05	4.70	3.04	3.49	4.24	2.71	3.29	4.92	3.14	3.69	4.37	2.82	3.39
		8	4.12	2.72	3.21	3.98	2.58	2.97	4.39	2.95	3.30	4.18	2.63	3.24	4.75	3.02	3.71	4.28	2.71	3.33
4		∞	4.03	1.99	3.21	3.92	1.92	2.88	4.64	**2.13**	**3.19**	4.08	1.97	2.97	4.63	2.20	3.40	4.12	2.05	3.06
		8	4.11	1.98	2.89	3.83	1.90	2.85	4.59	2.23	3.26	3.96	1.96	2.91	4.60	2.28	3.34	4.08	1.99	2.97
8		∞	3.91	1.93	3.07	3.51	1.89	2.85	4.17	2.26	3.22	3.79	1.91	2.93	4.19	2.20	3.23	3.87	1.97	2.99
		8	**3.51**	**1.88**	**2.87**	**3.39**	**1.86**	**2.79**	**3.84**	2.20	3.30	**3.63**	**1.88**	**2.82**	**4.17**	**2.14**	**3.22**	**3.74**	**1.93**	**2.90**
Baselines																				
LSTM		∞	6.23	8.14	6.96	6.23	7.45	6.92	6.97	8.28	7.66	5.72	7.11	6.81	6.32	7.95	7.81	5.59	6.46	7.21
		8	6.28	8.02	7.38	6.17	7.41	6.82	6.85	8.80	8.12	5.65	7.06	6.76	6.37	8.02	7.55	5.51	6.40	7.15
DCRNN		∞	6.54	7.54	7.22	6.13	7.25	6.83	6.78	8.47	8.13	5.85	7.21	6.87	6.65	8.43	7.87	5.85	6.42	7.17
		8	6.79	7.32	7.49	6.07	7.21	6.79	7.16	8.06	7.65	5.75	7.20	6.80	6.57	8.31	7.50	5.74	6.38	7.10
LAST			8.73	9.11	8.97	8.84	9.03	8.68	8.73	9.11	8.97	8.84	9.03	8.68	8.73	9.11	8.97	8.84	9.03	8.68
MEAN			8.20	9.00	8.63	7.75	8.18	7.71	8.89	9.63	8.68	11.32	13.17	9.83	12.51	15.24	11.26	13.46	13.70	9.76

Table 5: Statistics of application iteration times and prediction errors. The table lists the maximum and minimum iteration times (in nanoseconds), the standard deviation of iteration times, as well as the MAPE and RMSE of the SMART predictions. Despite the significantly higher iteration time variability under random placement, SMART maintains consistent accuracy across both placement strategies for all applications.

		Max iter. time (ns)	Min iter. time (ns)	Std.	SMART MAPE	SMART RMSE
Cont	MILC	1582600.75	1549385.13	3089.45	3.51	0.00247
	LAMMPS	1539606.96	1036303.24	130891.07	1.88	0.00122
	NN	322073.02	161752.20	19313.22	2.87	0.00272
Rand	MILC	2173786.95	1543215.38	100769.42	3.39	0.00259
	LAMMPS	6561415.11	1252410.89	807331.12	1.86	0.00109
	NN	740699.14	163661.67	59688.04	2.79	0.00244

duces greater iteration time variability due to network interference compared with contiguous placement. These findings demonstrate the robustness of SMART in predicting application runtime across placement strategies.

Discussion. Achieving 100% accuracy in practice is unrealistic, particularly in highly dynamic networking environments of large-scale systems. In a related study, researchers demonstrated that a hybrid PDES using a MEAN-based surrogate model could achieve comparable simulation performance (Cruz-Camacho et al. 2024).

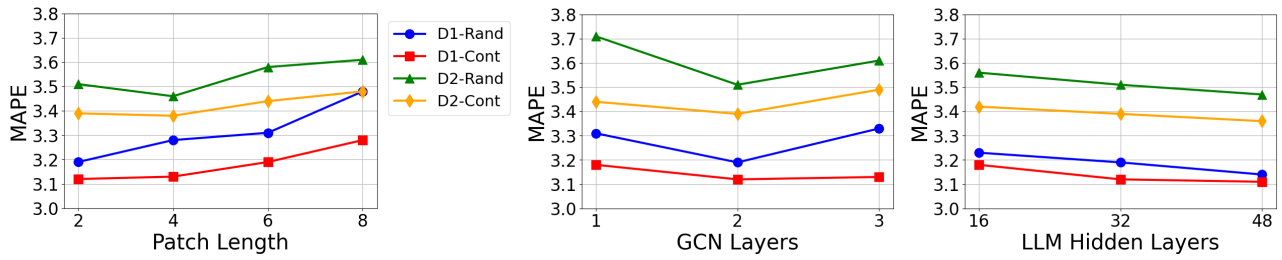
Our results show that SMART significantly outperforms MEAN across various scenarios. Prediction accuracy varies across different applications, with LAMMPS exhibiting higher precision than MILC. This discrepancy arises from their distinct sensitivities to network latency: LAMMPS is moderately sensitive, while MILC, with frequent global communication, is highly latency-sensitive. As a result, their sensitivity to application communication prediction also differs, affecting the accuracy of surrogate models. *Overall, our model outperforms in prediction accuracy by effectively incorporating domain-specific knowledge of both application characteristics and network architecture.*

Table 6: *Average inference time* (in seconds). Benchmarked on an 80-core ARM system with 2×A100 GPUs. All models, including SMART, achieve inference times significantly lower than the simulation per-iteration times (Table 7).

Model	Avg. Inference Time
SMART	0.5150
LLM-only	0.4158
GNN-only	0.0633
MEAN	0.00001
LAST	$<$ 0.00001
LSTM	0.0398
DCRNN	0.0459

Results on Inference Time

Table 6 shows the average inference times for all models, demonstrating that they are orders of magnitude faster than traditional PDES simulations. Inference times range from under 0.00001 seconds (LAST) to 0.5150 seconds (SMART). In contrast, Table 7 shows that PDES iteration times span from 8.09 to 243.48 seconds, depending on the applica-



(a) Effect of Patch Length (b) Effect of GCN Layers (c) Effect of LLM Hidden Layers

Figure 4: Effect of different hyper-parameters on MAPE for the different Datasets and node assignment combinations: (a) Path Length, (b) the number of GCN layers, and (c) the number of LLM hidden layer. The variations are not significant given that the MAPE varies between 3.1 and 3.8 the best baseline has a MAPE of 6.09.

Table 7: *Simulation time* per iteration (seconds) for each application in D_1 and D_2 under different job placements. All surrogate models (Table 6) are orders of magnitude faster, enabling integration with PDES.

Dataset	Application	Contiguous	Random
D_1	MILC	79.36	132.34
	LAMMPS	56.47	243.48
D_2	MILC	65.46	112.42
	LAMMPS	47.06	216.58
	NN	8.09	14.98

Table 8: MAPE results for GNN-only on D_1 for different T_{inGNN} and retrain frequencies F_T.

		Cont		Rand	
T_{inGNN}	F_T	MILC	LAMMPS	MILC	LAMMPS
2	∞	3.68	2.14	3.81	4.78
	8	3.46	2.05	3.58	4.59
4	∞	4.15	2.31	3.98	4.74
	8	3.86	2.26	3.70	4.65
8	∞	4.30	2.36	4.96	4.76
	8	3.95	2.30	4.56	4.64

tion and job placement. Even the slowest surrogate model (SMART) requires less than 1% of the shortest PDES iteration time and under 0.2% of the longest, making real-time use feasible. These results confirm the surrogate models are lightweight and suitable for integration into time-sensitive simulation workflows.

Discussion. A practical surrogate model must execute efficiently to benefit large-scale simulations. While traditional simulations can take hours or days to complete, SMART performs inference in just 0.5150 seconds per step, making hybrid simulation both efficient and feasible.

Ablation Study

We evaluate the contributions of the two main components of SMART: the GNN encoder and the Time-LLM module.

First, the GNN-only variant, which removes the LLM component and relies solely on the GNN encoder and temporal transformer, shows significantly degraded performance on D_1 (Table 8), indicating that LLM plays a critical role in capturing temporal patterns. Accuracy also improves when using a shorter online tuning window ($F_T = 8$). Table 9 shows the MAPE results for the GNN-only model on dataset D_2. This model removes the LLM component from SMART. In this dataset, the results are consistent with our findings for dataset D_1. The GNN-only model performs worse than the full SMART model. That implies that the LLM component is a critical component for SMART.

Table 9: MAPE results for GNN-only in D_2 for different T_{inGNN} and F_T, reorganized for single-column layout.

		Cont			Rand		
T_{inGNN}	F_T	MILC	LAMMPS	NN	MILC	LAMMPS	NN
2	∞	4.26	5.46	3.60	3.93	4.88	3.51
	8	3.85	4.91	3.64	3.69	4.68	3.44
4	∞	4.63	5.43	4.14	4.11	4.68	3.88
	8	4.41	5.57	3.80	3.82	4.59	3.84
8	∞	4.56	5.28	4.29	5.24	4.80	4.01
	8	4.27	5.24	4.37	4.82	4.68	3.89

Table 10: MAPE results for LLM-only for different T_{inLLM} for the simulation in D_1.

	Cont		Rand	
	MILC	LAMMPS	MILC	LAMMPS
$T_{inLLM} = 2$	4.23	2.77	4.24	2.77
$T_{inLLM} = 4$	4.36	1.98	4.33	1.90
$T_{inLLM} = 8$	4.02	1.94	3.97	1.91

Next, the LLM-only variant excludes the GNN and uses only temporal input to predict future iteration times based on a look-back window T_{inLLM}. This model also underperforms SMART on D_1 (Table 10), underscoring the importance of spatial features. Longer input windows generally lead to better accuracy by allowing the model to leverage more historical data. Table 11 presents the MAPE results for the LLM-only model on dataset D_2. The trend is similar to those ones on dataset D_1. The LLM-only model relies solely on the LLM module to predict future iteration times and also performs worse than the full SMART model as expected. This indicates that the GNN-based component is needed to capture spatial dependencies.

Table 11: MAPE results for LLM-only in D_2 for different T_{inLLM}.

	Cont			Rand		
	MILC	LAMMPS	NN	MILC	LAMMPS	NN
$T_{inLLM} = 2$	4.29	2.75	3.51	4.32	2.68	3.51
$T_{inLLM} = 4$	4.48	2.08	3.48	4.51	1.97	3.44
$T_{inLLM} = 8$	4.09	1.93	3.36	4.13	1.94	3.22

Table 12: Runtime prediction MAPE (%) on D2-cont when replacing SMART 's Time-LLM with Autoformer or DLinear.

Model Variant	MILC	LAMMPS	NN
SMART (Time-LLM)	**3.91**	1.93	**3.07**
GNN + Autoformer	4.22	**1.84**	3.68
GNN + DLinear	4.39	1.95	4.11
GNN + PatchTST	4.05	1.91	3.15

Parameter or Hyper-parameter variations

We analyze the impact of varying key hyperparameters to better understand their effect on prediciton accuracy. We observe only minor variations across hyperparameter settings, suggesting that SMART is robust and does not require extensive tuning to achieve strong performance.

Effect of Patch Length: Figure 4a illustrates the results (MAPE) produced by SMART when we vary the patch length of its LLM component. As observed, increasing the patch length slightly degrades the accuracy. This could be due to the lower granularity in capturing the temporal variations.

Varying GCN Layers: Figure 4b shows the impact of varying the number of GCN layers on the produced MAPE by SMART. Increasing the number of layers beyond two does not lead to a significant accuracy gain. In fact, deeper GCNs may introduce over-smoothing effects that hinders performance. This also implies that a surrogate model only needs a local structure from the network topology to predict application run time accurately in the Dragonfly systems.

Varying LLM hidden layers: The effect of the variation of the number of hidden layers in the LLM model is shown in Figure 4c. The variation does not have significant effect on the MAPE.

Temporal Model Comparisons

We evaluate the role of SMART 's Time-LLM by replacing it with three widely used time-series models: Autoformer, DLinear and PatchTST. All models used the same prompt-based input (T_{in} = 8) and fixed GNN encoding ($T_{in,GNN}$ = 2). Results on the D2-cont dataset are reported in Table 12 using mean absolute percentage error (MAPE).

SMART with Time-LLM outperforms both alternatives on MILC and NN workloads, while Autoformer achieves slightly better accuracy on LAMMPS. These results suggest that the LLM-based architecture in SMART provides more consistent performance across diverse workload behaviors, especially those with bursty or nonstationary iteration patterns.

Conclusion

We presented SMART, a domain-aware surrogate model designed to accelerate parallel discrete event simulation for Dragonfly interconnects. By integrating GNNs to model the hierarchical topology of Dragonfly networks and LLMs to capture temporal traffic dynamics, SMART effectively learns both the spatial structure and temporal behavior intrinsic to Dragonfly systems. Evaluated on 1,056-node Dragonfly simulations, SMART consistently outperforms baseline methods in prediction accuracy and delivers sub-second inference, reducing simulation time from hours to seconds.

Although this study centers on Dragonfly systems, the modular architecture of SMART allows for straightforward adaptation to other network topologies. For instance, applying SMART to alternative architectures like Fat Tree requires only retraining with new graph-structured inputs, without altering the core model. In future work, we plan to extend the model's forecasting horizon and explore its generalization across network types.

Acknowledgment

This work was supported in part by the U.S. Department of Energy under Contract No. DE-SC0024271 and by the U.S. National Science Foundation under Grants OAC-2402901 and CCF-2515009. The authors thank Kevin Brown and Rob Ross at Argonne National Laboratory and Chris Carothers at RPI for their helpful feedback and discussions.

References

2024. LAMMPS Benchmark [Online; accessed 11/13/2024]. https://www.lammps.org/bench.html.

2024. MILC Benchmark [Online; accessed 11/13/2024]. https://web.physics.utah.edu/~detar/milc/.

A., M. S.; and C., A. R. 1979. ANALYSIS OF FREEWAY TRAFFIC TIME-SERIES DATA BY USING BOX-JENKINS TECHNIQUES. *TRR*.

Carothers, C.; and et al. 2002. ROSS: A high-performance, low-memory, modular Time Warp system. *JPDC*, 62: 1648–1669.

Chen, C.; Oliveira, G.; Noghabi, H. S.; and Sylvain, T. 2024. LLM-TS Integrator: Integrating LLM for Enhanced Time Series Modeling. arXiv:2410.16489.

CODES. 2025. CODES on GitHub. https://github.com/codes-org/codes/.

Cruz-Camacho, E.; Brown, K.; Wang, X.; Xu, X.; Shu, K.; Lan, Z.; Ross, B.; and Carothers, C. 2024. Hybrid PDES Simulation of HPC Networks Using Zombie Packets. *ACM Trans. Model. Comput. Simul.*

Das, A.; Kong, W.; Sen, R.; and Zhou, Y. 2024. A decoder-only foundation model for time-series forecasting. arXiv:2310.10688.

Ferriol-Galmés, M.; Paillisse, J.; Suárez-Varela, J.; Rusek, K.; Xiao, S.; Shi, X.; Cheng, X.; Barlet-Ros, P.; and Cabellos-Aparicio, A. 2023. RouteNet-Fermi: Network modeling with graph neural networks. *IEEE/ACM transactions on networking*, 31(6): 3080–3095.

Fujimoto, R. M. 1990. Parallel discrete event simulation. *Commun. ACM*, 33(10): 30–53.

Garza, A.; Challu, C.; and Mergenthaler-Canseco, M. 2024. TimeGPT-1. arXiv:2310.03589.

Hamilton, W. L.; Ying, R.; and Leskovec, J. 2018. Inductive Representation Learning on Large Graphs. arXiv:1706.02216.

Hochreiter, S.; and et al."urgen. 1997. Long Short-Term Memory. *Neural Comput.*, 9(8): 1735–1780.

Jin, M.; Wang, S.; Ma, L.; Chu, Z.; Zhang, J. Y.; Shi, X.; Chen, P.-Y.; Liang, Y.; Li, Y.-F.; Pan, S.; and Wen, Q. 2024. Time-LLM: Time Series Forecasting by Reprogramming Large Language Models. arXiv:2310.01728.

Keahey, K.; Anderson, J.; Zhen, Z.; Riteau, P.; Ruth, P.; Stanzione, D.; Cevik, M.; Colleran, J.; Gunawi, H. S.; Hammock, C.; Mambretti, J.; Barnes, A.; Halbach, F.; Rocha, A.; and Stubbs, J. 2020. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association.

Kim, J.; and et al. 2008. Technology-Driven, Highly-Scalable Dragonfly Topology. In *2008 ISCA*, 77–88.

Kipf, T. N.; and Welling, M. 2017. Semi-Supervised Classification with Graph Convolutional Networks. arXiv:1609.02907.

LCRC. 2024. *Bebop*.

Liu, J.; Tang, F.; Zheng, Z.; Liu, H.; Hou, X.; Chen, L.; Gao, M.; Yu, J.; and Zhu, Y. 2024. Practical Network Modeling Using Weak Supervision Signals for Human-Centric Networking in Metaverse. *IEEE Journal on Selected Areas in Communications*, 42(3): 680–693.

Mubarak, M.; Carothers, C. D.; Ross, R. B.; and Carns, P. 2017. Enabling parallel simulation of large-scale HPC network systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(1): 87–100.

ORNL. 2022. Frontier.

Rizzini, P. L. 2024. *Predictive Modeling of Application Runtime in Dragonfly Systems*. Master's thesis, University of Illinois Chicago.

Rossi, E.; Chamberlain, B.; Frasca, F.; Eynard, D.; Monti, F.; and Bronstein, M. 2020. Temporal Graph Networks for Deep Learning on Dynamic Graphs. arXiv:2006.10637.

Rumelhart, D. E.; and et al. 1987. *Learning Internal Representations by Error Propagation*, 318–362.

Shiblee, M.; and et al. 2009. Time Series Prediction with Multilayer Perceptron (MLP): A New Generalized Error Based Approach. In Köppen, M.; Kasabov, N.; and Coghill, G., eds., *Advances in Neuro-Information Processing*, 37–44. ISBN 978-3-642-03040-6.

Stevens, R.; and et al. 2019. Aurora: Argonne's next-generation exascale supercomputer. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States).

Suárez-Varela, J.; Almasan, P.; Ferriol-Galmés, M.; Rusek, K.; Geyer, F.; Cheng, X.; Shi, X.; Xiao, S.; Scarselli, F.; Cabellos-Aparicio, A.; et al. 2022. Graph neural networks for communication networks: Context, use cases and opportunities. *IEEE network*, 37(3): 146–153.

top500.org. 2024. Top500 list.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L.; and Polosukhin, I. 2023. Attention Is All You Need. arXiv:1706.03762.

Veličković, P.; Cucurull, G.; Casanova, A.; Romero, A.; Liò, P.; and Bengio, Y. 2018. Graph Attention Networks. arXiv:1710.10903.

Wang, M.; Hui, L.; Cui, Y.; Liang, R.; and Liu, Z. 2022. xnet: Improving expressiveness and granularity for network modeling with graph neural networks. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, 2028–2037. IEEE.

Wang, X.; Mubarak, M.; Kang, Y.; Ross, R. B.; and Lan, Z. 2020. Union: An Automatic Workload Manager for Accelerating Network Simulation. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 821–830.

Wibawa, A.; and et al. 2022. Time-series analysis with smoothed Convolutional Neural Network. *Journal of Big Data*, 9: 44.

Wolfe, N.; Mubarak, M.; Jain, N.; Domke, J.; Bhatele, A.; Carothers, C. D.; and Ross, R. B. 2017. Preliminary performance analysis of multi-rail fat-tree networks. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 258–261. IEEE.

Yang, X.; Jenkins, J.; Mubarak, M.; Ross, R. B.; and Lan, Z. 2016a. Watch out for the bully! job interference study on dragonfly network. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 750–760. IEEE.

Yang, X.; Jenkins, J.; Mubarak, M.; Wang, X.; Ross, R. B.; and Lan, Z. 2016b. Study of intra-and interjob interference on torus networks. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, 239–246. IEEE.

Zhu, D.; and et al. 2014. Developing A High Performance Computing (Hpc) Based Hydrological Modelling Framework To Support Extreme Weather Impact Studies.