# Identifying Degree and Sources of Non-Determinism in MPI Applications Via Graph Kernels

Dylan Chapp, Nigel Tan, Sanjukta Bhowmick, and Michela Taufer, *Senior Member, IEEE*

**Abstract**—As the scientific community prepares to deploy an increasingly complex and diverse set of applications on exascale platforms, the need to assess reproducibility of simulations and identify the root causes of reproducibility failures increases correspondingly. One of the greatest challenges facing reproducibility issues at exascale is the inherent non-determinism at the level of inter-process communication. The use of non-deterministic communication constructs is necessary to boost performance, but communication non-determinism can also hamper software correctness and result reproducibility. To address this challenge, we propose a software framework for identifying the percentage and sources of communication non-determinism. We model parallel executions as directed graphs and leverage graph kernels to characterize run-to-run variations in inter-process communication. We demonstrate the effectiveness of graph kernel similarity as a proxy for non-determinism, by showing that these kernels can quantify the type and degree of non-determinism present in communication patterns. To demonstrate our framework's ability to link and quantify runtime non-determinism to root sources, demonstrate with present for an adaptive mesh refinement application, where our framework automatically quantifies the impact of function calls on non-determinism, and a Monte Carlo application, where our framework automatically quantifies the impact of parameter configurations on non-determinism.

**Index Terms**—Non-determinism, reproducibility, debugging, trace analysis, graph similarity

## 1 Introduction

Scientific applications commonly overlap multiple asynchronous communication routines to achieve scalability on high performance computing (HPC) platforms. In practice, this overlap requires the use of non-deterministic communication patterns. If improperly managed, communication non-determinism leads to costly bugs [1], [2] and hampers reproducibility of simulations [3], [4]. As scientific applications change over time (e.g., adding new features or running at larger scales), the impacts of non-determinism are exacerbated and users' ability to reason about *root causes* of non-determinism is inhibited [5], [6]. By root causes, we refer to patterns of non-deterministic communication constructs in MPI applications. These patterns are commonplace in applications with irregular communication (e.g., bioinformatics, graph analytics) in which processes may not know the number, the order, or the source of the messages they will receive. There is a critical need to enhance scientists' ability to comprehend non-determinism in these kinds of applications as the HPC community moves towards exascale.

- *Dylan Chapp, Nigel Tan, and Michela Taufer are with the University of Tennessee at Knoxville, Knoxville, TN 37996 USA. E-mail: dylanchapp@gmail.com, ntan1@vols.utk.edu, mtaufer@utk.edu.*
- *Sanjukta Bhowmick is with the University of North Texas, Denton, TX 76203 USA. E-mail: sanjukta.bhowmick1@gmail.com.*

*Manuscript received 31 July 2020; revised 5 Apr. 2021; accepted 1 May 2021. Date of publication 18 May 2021; date of current version 3 June 2021. (Corresponding author: Michela Taufer.) Recommended for acceptance by K. Mohror. Digital Object Identifier no. 10.1109/TPDS.2021.3081530*

To this end, we present a software framework for quantifying and characterizing communication non-determinism through the lens of graph similarity. Our framework enables construction of graph models of executions whereby we leverage graph kernels to quantify subtle variations in non-deterministic communication patterns. These are precisely the kind of variations that may indicate underlying software correctness and scientific reproducibility issues in production applications.

While the scope of our work is not to resolve software correctness and scientific reproducibility, our framework provides scientists with valuable insights to link runtime manifestations of non-determinism to their root causes in source code, without *a priori* knowledge of the application's communication patterns. These insights are a prerequisite to address correctness and reproducibility. Our framework targets communication patterns composed of point-to-point MPI communication constructs. These constructs are typical of non-deterministic application's communication patterns, as presented in [1], [2], [7]. Furthermore, characterization of trends in how HPC applications use MPI features presented in [8] proves that the majority of MPI applications with non-deterministic patterns use point-to-point communication constructs, guaranteeing the broader impact of our framework across scientific domains. Thus, the value proposition of our work is that, supported by our framework, scientists can ultimately bootstrap their process for diagnosing non-deterministic bugs or addressing reproducibility failures reducing the labor-intensive manual inspection of complex code. Our contributions in this paper are:

- To devise a novel extension of event graph modeling in which we transform graph kernels to become a quantitative proxy for communication non-determinism of MPI applications.
- To build a software framework that leverages our application of graph kernels to event graphs, thereby linking runtime manifestations of non-determinism to root causes in source code.
- To empirically validate the effectiveness of our software framework against a suite of four communication patterns of increasing complexity and realism extracted from HPC benchmark suites.
- To apply our software framework to two applications, an adaptive mesh refinement code and a Monte Carlo simulation code, to automatically identify non-determination embedded into function calls and configuration parameters respectively, without a manual analysis of the code or the use of costly monitoring and recording tools.

The remainder of the paper is structured as follows. In Section 2 we provide two examples of how non-determinism negatively impacts software correctness and scientific reproducibility. In Section 3 we describe our framework for characterizing non-determinism in MPI applications and demonstrate our framework's ability to quantify the degree of non-determinism present in a given communication pattern. In Section 4 we present two case studies on non-deterministic MPI application where we apply our framework to automatically identify sources of non-determinism. We review related work in Section 5. Finally, we summarize our findings in Section 6.

## 2 Impact of Non-Determinism

Ensuring correctness and reproducibility of non-deterministic applications is a broad problem that requires both automatic tools to identify the root causes of non-determinism as well as manual contributions from the application developers to address these causes while preserving the functionality of the application. In this paper we address the first item by developing a software framework that can identify the degree and sources of non-determinism in large-scale MPI applications, reducing the responsibility from manual analyses or the need to use monitoring and recording tools.

Large-scale MPI applications typically make opportunistic decisions at runtime on the order in which processes exchange data in order to improve their performance. Consequently, non-deterministic communication patterns have become a feature of these scientific applications on HPC systems. Due to this inherent non-determinism, the performance gains come at the cost of ensuring software correctness and scientific reproducibility. In other words, non-determinism renders it difficult for developers to track the variations of program flow for debugging, which is a prerequisite of software correctness. It is also challenging to reproduce the results over repeated runs, thus making it hard to trust the scientific outputs. Below, we list two real-world cases demonstrate the critical need to characterize the degree and localize the source of non-determinism in HPC applications on future exascale resources.

*Impact on Software Correctness.* A recent DOE report [2] highlights how "non-determinism control" is needed to identify non-deterministic bugs and ensure correctness of HPC software in the exascale era. As reported in an HPC debugging case study [1], a team of scientists were impeded by a non-deterministic bug that caused intermittent hangs after multiple hours of execution. The bug was located in the linear algebra package HYPRE 2.10.1 [9] that was used by the scientists research application Diablo [10], a multi-physics finite-element code. This bug impeded the scientists research progress for 18 months, and more than 10,000 hours (approximately one year) of compute time were spent localizing and diagnosing the bug.

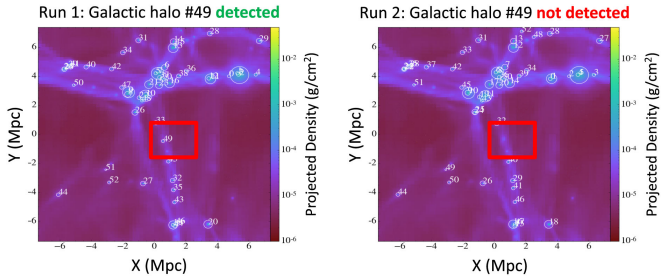



Fig. 1. Discrepancy between two runs of galaxy formation simulation using the Enzo adaptive mesh refinement code [17] due to non-determinism. Documented by Stodden *et al.* in [3].

*Impact on Scientific Reproducibility.* Many studies [3], [11], [12] have demonstrated that application non-determinism can have harmful impacts on the trustworthiness of simulation outcomes. As a consequence, in recent years, major HPC publication venues have mandated assessing reproducibility of submitted research [13], [14], and major HPC laboratories have prioritized investment in software tools [15], [16] with the explicit goal of characterizing, quantifying, and controlling non-determinism for the purpose of computational reproducibility.

A recent study [3] on replicability of computational science results shows how non-determinism impacts scientific reproducibility. In Fig. 1, we show an excerpt from that study that compares the output of two runs of a simulation of galaxy formation using the Enzo adaptive mesh refinement code [17]. In Run 1, a specific galactic halo (49) formed over the course of the simulation, while in a subsequent run (Run 2) that same galactic halo did not form. This type of discrepancy in outputs of simulations undermines the trustworthiness of scientific conclusions based upon them. Worse than the discrepancy itself is the fact that despite rigorous investigation in [3], the exact root causes of the discrepancy remain poorly understood.

## 3 Our Modeling Framework

We develop a framework to model and quantify non-determinism in MPI applications. Our framework allows scientists to characterize the source of non-determinism in a application without *a priori* knowledge of its communication patterns. The framework models non-determinism through three steps: *(Step 1)* execution-trace collection, *(Step 2)* event graph model construction, and *(Step 3)* event graph analysis.

Fig. 2 outlines the software and data components of the framework, its inputs (i.e., the non-deterministic MPI applications and the application's traces), and its outputs (i.e.,

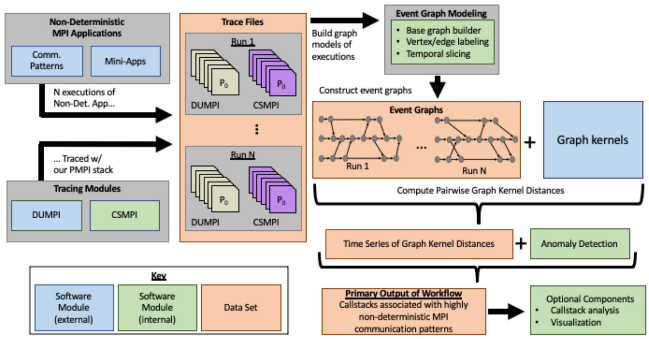



Fig. 2. Our non-determinism modeling framework including execution trace collection, event graph model construction, and event graph analysis enabling anomaly detection, callstack analysis, and visualization.

identification of code regions and configuration parameters affecting non-determinism).

### 3.1 Step 1: Execution-Trace Collection

As a *first step* to modeling non-determinism, our framework captures traces of multiple executions of a target non-deterministic application via two tracing modules: CSMPI and DUMPI [18], [19].

CSMPI is a tracing module that we implement to capture callstacks associated with MPI function calls. By collecting callstacks of communication events, the framework can associate periods of run-to-run variability in message order across multiple executions with paths in the application's call-graph. In doing so, we effectively link the runtime manifestation of non-determinism with a set of potential root causes related to specific patterns of MPI calls in the application's source code.

DUMPI generates an unambiguous record of the message order (i.e., the specific interleaving of message sends and receives on each MPI process during the traced execution). We select DUMPI because of its low overhead; it does not impose undue probe effects [20] on non-deterministic applications.

The framework uses P$^N$MPI [21], [22], a tool for composing PMPI modules, to "stack" CSMPI and DUMPI so that they can be linked with target applications as if they were a single module as shown in Fig. 3.

For each execution of the target application, the framework generates one CSMPI trace file and one DUMPI trace file per MPI process (or rank). The trace files are subsequently ingested by the event graph constructor to reconstruct the execution's message order, as described in Section 3.2. While not all the individual modules are novel in themselves, how we combine them into an extendable, modular framework is in

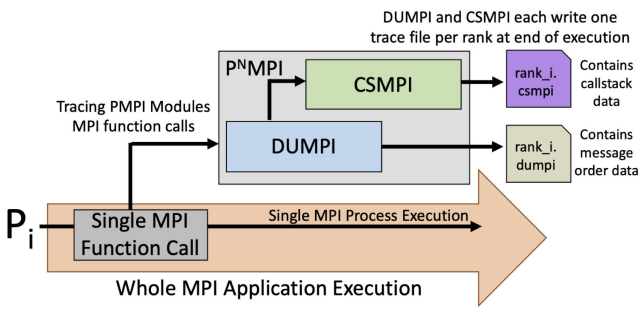



Fig. 3. Our P$^N$MPI-based tracing stack including DUMPI and CSMPI stacked as a single module.

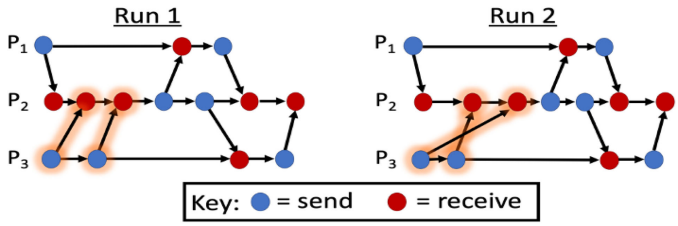



Fig. 4. Example of two event graphs of non-deterministic executions with differences in message order from Run 1 to Run 2.

itself a powerful contribution. Our modular design allows us to extend our tracing stack as needed to encompass other, new functionality, such as adding a noise-injection module [1] to explore the effects of message order perturbation, or adding a module to trace OpenMP runtime events for hybrid MPI +OpenMP applications.

### 3.2 Step 2: Event Graph Model Construction

In the *second step*, our framework models the execution of a non-deterministic application as a directed acyclic graph (DAG) using the trace files generated in the first step. Here the vertices represent point-to-point communication events such as message sends and message receives, and directed edges represent happens-before relationships [23] between those events. Models of inter-process communication of this form are typically referred to as event graphs [24], and have their roots in early visual debugging tools for MPI applications [25].

Fig. 4 shows a simple example of two event graphs representing two runs of the same application, highlighting how the difference in message order is represented in the event graph structure. In this figure, we consider a simulation across three processors, each represented by a row in the graph. Processor 1 is the top row, Processor 2 is the middle row, and Processor 3 is the bottom row. The vertices in each row represent the send and receive MPI functions in each processor. Two vertices are connected if they are consecutive operations in the same processor or sent/received message to another processors.

During Run 1 the *first event* in Processor 3 sends a message that is received as the *second event* in Processor 2, and the *second event* in Processor 3 sends a message that is received as the *third event* in Processor 2. On the other hand, during Run 2, the *first event* in Processor 3 sends a message that is received as the *third event* in Processor 2, and the *second event* in Processor 3 sends a message that is received as the *second event* in Processor 2. Thus the two different runs result in event graphs of different structures.

While modelling executions with graphs is not new in itself, our use of the graphs has two aspects of novelty. *First*, we use graph kernel similarity between event graphs (or subgraphs thereof) to quantify the degree of non-determinism in the underlying application's message order (Section 3.2.1). *Second*, we enrich event graphs with vertex label data with semantics relevant to communication non-determinism (e.g., the callstacks traced by CSMPI) (Section 3.2.2).

#### 3.2.1 Non-Determinism Quantification via Graph Kernels

Graph kernels [26], [27] are a family of methods for measuring the structural similarity of graphs. We leverage graph

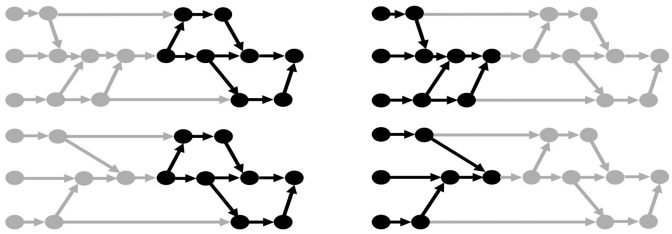



Fig. 5. General intuition for graph kernel similarity. Matching sub-structures (black subgraph in Fig. 5a) increase the similarity score of the pair of input graphs. Non-matching substructures (black subgraph in Fig. 5b) do not increase the similarity score.

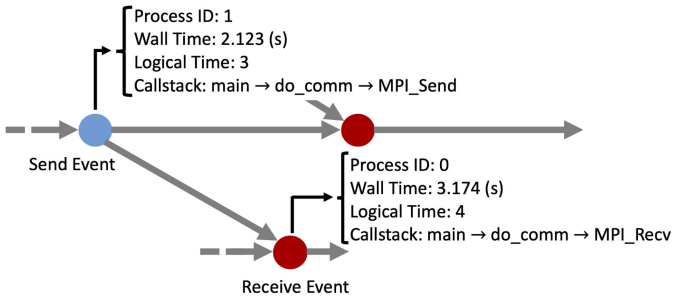



Fig. 6. Example event graph model with vertices representing MPI communication events, directed edges representing happens-before relations between communication events, and callstack vertex labels enabling our framework to link runtime non-determinism to root causes in source code.

kernels to quantify (dis)similarity of event graphs that model multiple runs of a target application, thereby quantifying the degree to which non-determinism manifests in the application.

*Overview of Graph Kernels.* Formally, a graph kernel is a function $K : \mathcal{G} \times \mathcal{G} \rightarrow \mathbb{R}^+ \cup \{0\}$ where $\mathcal{G}$ denotes a set of graphs. A graph kernel is defined as $K(G, G') = \langle \phi(G), \phi(G') \rangle$ where $G, G' \in \mathcal{G}$ and $\phi(G)$ denotes the *embedding* of $G$ in a specific kind of vector space (i.e., a Reproducing Kernel Hilbert Space [26]). In this formulation, the kernel similarity $K(G, G')$ is the value of the inner product of the embeddings of $G$ and $G'$ in that vector space. Intuitively, it suffices to treat a graph kernel as a function that counts matching substructures of some kind (e.g., subtrees) of two input graphs, as shown in Fig. 5, mapping pairs of graphs to scalars that quantify how similar they are.

If the graph kernel is positive definite, as the ones we use for our modeling are, the graph kernel similarity score induces a metric on the underlying space of graphs: effectively, a "graph kernel distance" that describes how far apart a pair of graphs are from each other [28]. For a graph kernel $K$ and a pair of input graphs $G, G'$, the kernel distance between $G$ and $G'$ with respect to $K$ is given by

$$d_K(G, G') = \sqrt{K(G, G) + K(G', G') - 2K(G, G')},$$

where $K(G, G)$ and $K(G', G')$ represent the self-similarities of $G$ and $G'$ respectively and $K(G, G')$ represents their cross-similarity. This is a standard form of computing the kernel distance as given in non-alignment-based graph similarity literature such as [28].

*Graph Kernels for Modeling Non-Determinism.* We use graph kernels to quantify the similarity between communication patterns for a pair of event graphs that model a pair of executions. The graph kernel similarity between the event graphs naturally translates to measurement of the similarity of the communication events in the executions the event graphs model. This property effectively provides a means of quantifying the run-to-run differences in non-deterministic communication patterns.

We use the Weisfeiler-Lehman Subtree-Pattern (WLST) graph kernel [27], which embeds information about the graph structure surrounding each vertex via a user-selected number of rounds of *vertex refinement* [27], [29], denoted typically by $h$. During vertex refinement, the label of each vertex is replaced by a hash of the concatenation of a sorted multiset of its neighbors' vertex labels. By varying $h$, information about the graph structure of an increasingly large neighborhood around each vertex is incorporated into that vertex's label. The WLST kernel similarity is then simply defined as the sum of vertex-histogram similarities over the $h$ rounds of vertex refinement. While there are many different graph kernels [30], we select the WLST kernel for two reasons. First, the asymptotic time complexity of the WLST graph kernel algorithm is $O(Nh)$ where $N$ is the size of the vertex set of the largest input graph and $h$ is the number of rounds of vertex refinement, where typically $h \ll N$. The event graphs can be extremely large due to modeling long-running executions on many processes, so it is critical that the modeling steps should be scalable. Second, the WLST kernel is particularly appropriate for quantifying similarity of event graphs due to its ability to use arbitrary label data during vertex refinement, in contrast with other graph kernels such as the Random Walk kernel [31] that *only* operate on graph structure.

### 3.2.2 Event Graph Labeling

We enrich our event graph construction by associating labels to vertices that have semantic relevance for communication non-determinism, as shown in Fig. 6. We use two types of labeling that serve different functions: labels to identify type of non-determinism and labels to identify causes of non-determinism.

*Labels to Identify Type of Non-Determinism.* Vertex labels are added to the graph to provide the WLST kernel with additional relevant information to assist in distinguishing between event graphs with similar structure but different underlying non-determinism. The extra information that these vertex labels provide to the WLST kernel, as opposed to the bare graph structure itself, are crucial when the differences in message order from run to run are subtle. Our framework currently supports three kinds of vertex labels: process IDs, logical timestamps, and logical ticks.

Process ID vertex labels are simply the MPI rank of the process in which the event occurred, enabling the WLST kernel to detect when a process's set of neighbors varies from run to run due to non-determinism.

Logical timestamps, such as scalar Lamport time-stamps [23], embed information about patterns of concurrency in the event graph. This enables the WLST kernel to take into account the history of data flow from one process to another over the course of the execution.

Logical tick labels are derived from logical timestamps in the following way. Let $LTS(v)$ denote the logical timestamp of a vertex $v$, let $pred_{local}(v)$ denote the vertex immediately preceding $v$ in the program order of the process to which $v$ belongs, and let $pred_{remote}(v)$ denote the predecessor of $v$ in the program order of a remote process (i.e., in the case where $v$ represents a receive). Then the logical tick of $v$ is given by $max\{LTS(v) - LTS(pred_{local}(v)), LTS(v) - LTS(pred_{remote}(v))\}$. By using logical ticks as vertex labels, the WLST kernel can detect run-to-run difference in terms of patterns of message lateness.

*Labels to Identify Causes of Non-Determinism.* While analyzing the graph model, a vertex labeling scheme is used to link runtime non-determinism to root causes in application source code. The framework accomplishes this task by way of the callstack labels (i.e., the chain of function calls that terminates in the MPI function call that vertex represents). In other words, each callstack represents a path in the function call graph of the application. Thus, the aggregate collection of callstacks associated with regions of interest in event graphs (e.g., regions of particularly low similarity across multiple executions) represents an identification of the "hot paths" associated with non-deterministic communication patterns.

### 3.3 Step 3: Event Graph Analysis

In the *final step*, the event graph analysis allows scientists to quantify non-determinism from multiple executions of an MPI application, without any knowledge of the application's MPI communication patterns. To validate the suitability of graph kernel distances as the proxy for non-determinism during the analysis, we consider four diverse communication patterns as described in Section 3.3.1, we generate event graphs for these communication patterns with different, known percentages of non-deterministic communication volume (Section 3.3.2), and we empirically validate that graph kernel similarity is indeed a suitable quantitative proxy for the amount of non-determinism in the application behaviour (Section 3.3.3).

#### 3.3.1 Defining a Set of Communication Patterns

We consider four diverse communication patterns: the simple but ubiquitous Message Race, the receiver-side and sender-side non-deterministic patterns in the Algebraic Multigrid 2013 Benchmark (AMG2013) [32], the non-blocking MPI 2-dimensional grid communication pattern (MCB Grid) in the Monte Carlo Benchmark [33], and the Unstructured Mesh pattern [34] with its randomized process topology.

*Message Race.* When executed on $N$ processes this pattern consists of $N - 1$ sender processes sending a single message to a root process that receives them in arbitrary order. This communication pattern can be viewed as the atomic communication motif from which all other non-deterministic communication patterns that exhibit receiver-side non-determinism are composed.

*AMG2013.* The AMG2013 communication pattern is extracted from the proxy application of the same name in the CORAL Benchmark Suite. We select this pattern due to its expression of both receiver-side and sender-side non-determinism, a property that has been highlighted in prior work on communication non-determinism [7]. The AMG2013 pattern exhibits non-determinism due to its use of non-blocking wildcard probes. When one of these probes matches with a message, a receive for that message is posted. While the receive itself is not a wildcard receive (i.e., the source parameter is a fixed value rather than `MPI_ANY_SOURCE`), the source parameters value is non-deterministic as a consequence of the wildcard probes non-determinism. Upon completion of the receive, a send is issued to the process whose message was originally matched by the wildcard probe. Ultimately, the way that non-determinism propagates from the probe to the receive and send is not obvious from simple inspection of the receive and send callsites.

*MCB Grid.* The MCB 2-dimensional grid communication pattern is extracted from the MCB proxy application in the CORAL Benchmark Suite. We select this pattern due to its prior use for evaluating non-determinism management tools [35]. The MCB Grid pattern exhibits non-determinism because it uses the non-blocking MPI matching function `MPI_Testsome`.

*Unstructured Mesh.* This communication pattern, extracted from the Chatterbug Communication Pattern Suite [34], exhibits non-determinism due to a randomized process topology, resulting in run-to-run variation in terms of which processes communicate with which others. This manifestation of non-determinism contrasts with the other three patterns in which only the order of messages varies from run to run.

Furthermore, we consider three distinct ways in which non-determinism can manifest in the four communication patterns described above:

- Receiver-side non-determinism (Recv): messages can be received by a process in a variable order.
- Sender-side non-determinism (Send): processes can send messages in a variable order.
- Process topology non-determinism (Topo): applications where the set of processes a given process communicates with can vary from run to run.

We refer to the first two types as *message non-determinism* and the third type as *topology non-determinism*. The MPI features responsible for non-determinism within these communication patterns are: `MPI_ANY_SOURCE` (Message Race and AMG2013), `MPI_Testsome` (MCB Grid), and `MPI_Waitany` (Unstructured Mesh). The communication patterns are summarized in Table 1.

#### 3.3.2 Generating Communication Pattern Event Graphs

We implement each communication pattern such that a user-specified fraction of the message volume is non-deterministic. In the case of Unstructured Mesh, both communication patterns and process topology are non-deterministic. We refer to this fraction as message non-determinism percentage or topology non-determinism percentage respectively.

As an example of *message non-determinism*, if we run the Message Race pattern with a non-determinism percentage of 50 percent, then half of the messages are received in a deterministic order while the remaining half are received in an arbitrary order that varies from run to run. In other words, only half of the messages actually race. As an example of *topology non-determinism*, applicable only to Unstructured Mesh, if

TABLE 1
Communication Pattern Non-Determinism Properties

| Comm. Pattern | Recv | Send | Topo | Source |
|---|---|---|---|---|
| Message Race | ✓ | × | × | MPI_ANY_SOURCE |
| AMG2013 | ✓ | ✓ | × | MPI_ANY_SOURCE |
| MCB Grid | ✓ | ✓ | × | MPI_Testsome |
| Unstructured Mesh | ✓ | ✓ | ✓ | MPI_Waitany + RNG |

*A ✓ indicates if a communication pattern expressing the non-determinism of a given kind is present, and an ×shows that type of non-determinism is absent. In the table* Recv *stands for receiver-side non-determinism;* Send *stands for sender-side non-determinism; and* Topo *stands for process topology non-determinism; and* RNG *stands for random number generation used to randomize process neighborhoods.*

we run Unstructured Mesh with a topology non-determinism percentage of 50 percent, then each process will determine half of its partners deterministically while the remaining half will vary from run to run.

For each configuration of each communication pattern (e.g., Message Race with message non-determinism percentages 10, 20 percent, . . . , 100 percent), we trace 100 36-process executions of the pattern and construct their corresponding event graphs. We tag each event graph with its message non-determinism percentage. For the event graphs that model executions of Unstructured Mesh, we also tag the graphs with the topology non-determinism percentage. We observe that the cost in runtime to construct each event graph from the traces is negligible compared to the cost of tracing the communication patterns. Table 2 summarizes the values of message non-determinism percentage and topology non-determinism percentage that we explore in this work.

### 3.3.3 Validating Graph Kernel Similarity

We empirically validate the hypothesis that graph kernel similarities can quantify communication non-determinism by measuring the effectiveness of different graph kernel-based models for predicting the message non-determinism percentage of event graphs. The first step of building these models is to create one similarity matrix per model, where each entry gives the similarity score of a pair of event graphs with respect to a particular configuration of the WLST kernel. For each communication pattern (e.g., Message Race), we consider 10 possible settings of message non-determinism percentage (i.e., as enumerated in Table 2), and for each setting we build 100 event graphs. Thus, we have a $1000 \times 1000$ matrix of similarity scores. We then train a support vector machine regression model (SVR) on the event graph embeddings in the similarity matrix. For the training process, we perform 10-fold cross-validation. The success of our validation lies in whether the SVR can accurately predict the message non-determinism percentage of previously unseen event graphs. Our graph kernel-based SVR models predict the numerical value of the underlying message non-determinism percentage parameter for event graphs in the testing set. Thus, the prediction error of an SVR model (as shown in Figs. 8, 9, 10, and 11) indicates the degree to which the true amount of non-determinism in the modeled communication pattern is either over-estimated or under-estimated.

TABLE 2
Configuration Parameters for Controlling Non-Determinism in Communication Patterns

| Configuration Parameter (%) | Values |
|---|---|
| Message Non-Determinism | 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 |
| Topology Non-Determinism | 0, 50, 100 |

TABLE 3
Parameter Space for the WLST Kernel

| Parameter | Values |
|---|---|
| Vertex Label | Process ID, Logical Time, Logical Tick |
| WL-Iterations | {2, 4, 6, 8, 10} |

*Graph Kernel Parameters.* The similarity scores are affected by two graph kernel parameters: (i) the choice of vertex label for the event graph and (ii) the number of WL-iterations performed. We use one of the following three types of vertex labels, as described in Section 3.2.2: (a) the process ID of the vertex; (b) the logical timestamp; or (c) logical tick. We use one of the following numbers of WL-iterations: 2, 4, 6, 8, or 10. Increasing the number of iterations incorporates more graph-structural context into each vertex, but at the cost of increased runtime. We summarize the space of WLST kernel parameters in Table 3. Each combination of parameters yields a separate similarity matrix that in turn determines the prediction accuracy of the SVR model. Thus, each communication pattern is associated with 15 = 3 (Vertex Labels) × 5 (WL-iterations) different similarity matrices.

*Message Race.* We first validate our framework on the Message Race pattern. We start with this communication pattern for two reasons. First, we can easily tune the message non-determinism percentage by varying the number of fixed-source versus wildcard receives that the root process posts, as shown in Listing 1. Second, this simple setting provides a baseline for the discriminatory power that can be expected of the WLST kernel when applied to more complex communication patterns. Fig. 7 shows two examples of Message Race where four incoming messages are received deterministically (i.e., in the same order from run to run), and four incoming messages are received non-deterministically (i.e., in arbitrary order from run to run).

**Listing 1.** Message Race Pseudocode: Non-Determinism Percentage Controlled by $N_{DET}$ and $N_{ND}$ Parameters

```
if ( rank == 0 ) {
  for ( i =0; i<num_DET; ++i ) {
    MPI_Recv (... , i , ... ) ;
  }
  for ( i =0; i<num_ND; ++i ) {
    MPI_Recv ( ... , MPI_ANY_SOURCE, ... ) ;
  }
} else {
  MPI_Send ( ... , 0 , ... ) ;
}
```

We present validation results in Figs. 8a (process ID vertex label), Fig. 8b (logical time vertex label), and Fig. 8c

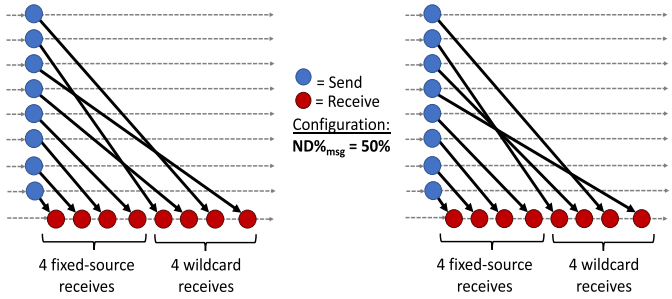



Fig. 7. Example of two event graphs for the message race pattern on nine processes with message non-determinism percentage 50% (i.e., four deterministic receives and four nondeterministic receives).

(logical tick vertex label). The three figures show the relative error of the SVR model predictions when the WLST kernel similarities are computed using different vertex labels (i.e., process ID, logical time, and logical tick). In each figure, each group of box plots on the $x$-axis represents the prediction errors for event graphs with a fixed true non-determinism percentage. Each box plot within a group represents the prediction errors when using the number of WL-iterations (i.e., 2, 4, 6, 8, and 10). In all cases, the $y$-axis represents the relative error in predicting the non-determinism percentage. In other words, lower values indicate more accurate predictions of the non-determinism percentage. The horizontal dashed line represents a relative error of 10 percent.

These figures highlight three features. *First*, the relative prediction error is low (i.e., smaller than 10 percent) across the space of WLST kernel parameters (i.e., choice of vertex label and number of WL-iterations), thus demonstrating that for Message Race, graph similarity metrics can indeed be used for quantifying non-determinism. *Second*, there are diminishing returns to increasing the number of WL-iterations, and the relative benefit of doing so depends on the kind of vertex labels applied to the event graph. *Third*, the largest prediction errors occur for event graphs with relatively low true non-determinism percentage. This phenomenon may occur because event graphs with relatively low, but different, message non-determinism percentage (e.g., one with 10 percent and one with 20 percent) are more similar to each other than those with high, but different, message non-determinism percentage (e.g., one with 80 percent and one with 90 percent). If the graphs are structurally more similar then mis-prediction is more likely.

*AMG2013*. Next in the order of complexity among the communication patterns, AMG2013 is a composition of message races where receiver-side non-determinism induces sender-side non-determinism. We run 100 36-process executions of the pattern per configuration given in Table 2 across the space of vertex labels and WL-iterations enumerated in Table 3. We show the relative non-determinism percentage prediction errors for AMG2013 in Figs. 9a (process id vertex label), Fig. 9b (logical time vertex label), and Fig. 9c (logical tick vertex label). The trends in the prediction errors are still similar to those observed for Message Race. Specifically, lower non-determinism percentage configurations have higher error rates. Larger numbers of WL-iterations provide diminishing returns in terms of accuracy improvements, including slight decreases in accuracy for high non-determinism percentage event graphs. The overall error rate remains below 10 percent, validating that WLST kernel similarity can effectively serve as a proxy for non-determinism in communication patterns consisting of compositions of message races.

*MCB Grid*. The MCB Grid communications use `MPI_Testsome` to complete a variable number of non-blocking receives per iteration; the communications issue sends in response to completion of receives. This results in a non-deterministic interleaving of sends and receives. For example, in one run, at iteration $i$ a process $p$ may complete one receive and issue one send in response, then at iteration $i+1$, $p$ may complete two receives and issue two sends. However, in a subsequent run, $p$ may complete two receives at iteration $i$ and one receive at iteration $i+1$. This behavior contrasts with that of Message Race and AMG2013 where the pattern of sends and receives is deterministic, despite the destinations and sources of those sends and receives being non-deterministic. Similarly, as for AMG2013, we run 100 36-process executions of the pattern per configuration given in Table 2. Figs. 10a (process id vertex label), Fig. 10b (logical time vertex label), and Fig. 10c (logical tick vertex label) show our framework's prediction errors for MCB Grid. We observe higher prediction errors for MCB Grid than for either Message Race or AMG2013, which we attribute to the more complex interleaving of receives and sends. We also note that our best prediction accuracy is

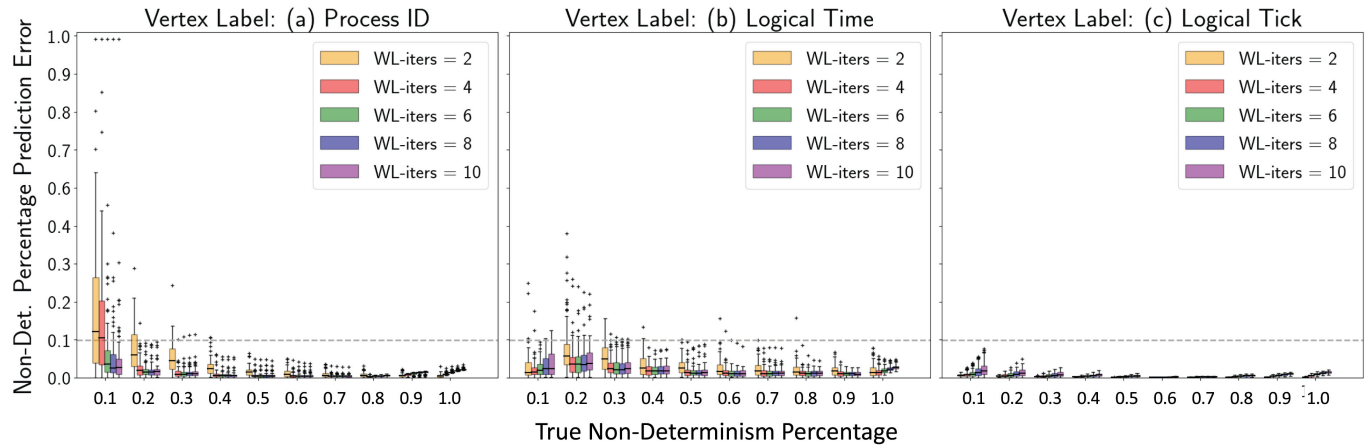



Fig. 8. Relative prediction errors for the non-determinism percentage on the Message Race communication pattern.

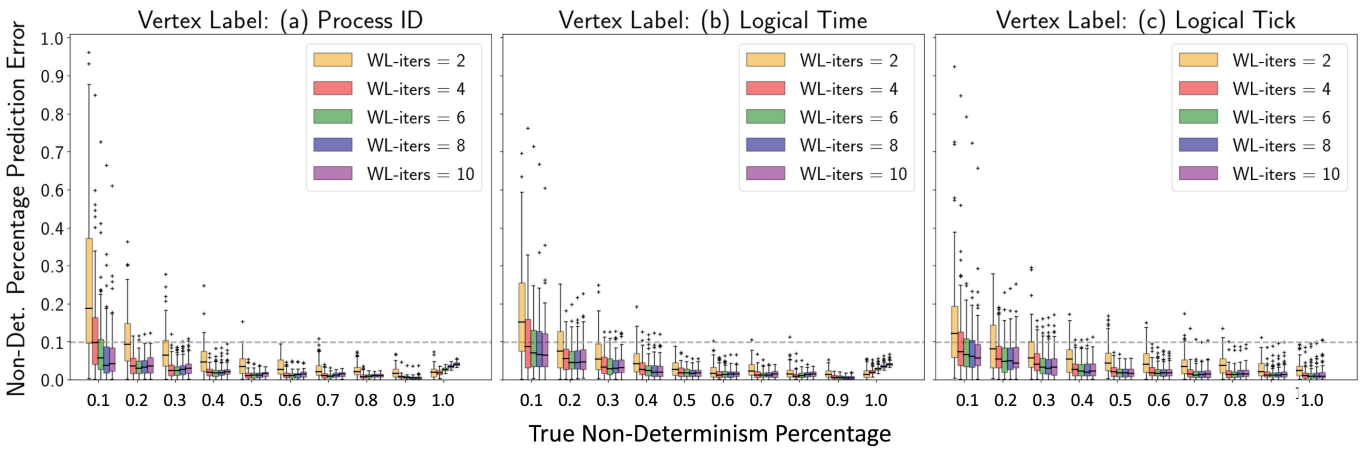



Fig. 9. Relative prediction errors for the non-determinism percentage on the AMG2013 communication pattern.

when using process ID vertex labels. This contrasts with results on Message Race and AMG2013 where logical time and logical tick vertex labels perform better. The reason for this phenomenon may be that characteristic patterns in the interleaving of receives and sends are being conveyed to the model by the propagation of process ID label values during WL-iterations.

*Unstructured Mesh.* The communication pattern of the Unstructured Mesh is the most complex of our communication patterns. The randomized nature of its process topology (i.e., the relation describing which processes may communicate with which others) presents a unique challenge. In other words, the Unstructured Mesh pattern includes interleaved non-deterministic sends and receives and non-deterministic process topology.

We show results on non-determinism percentage prediction in Figs. 11a, 11b, 11c, 11d, 11e, 11f, 11g, 11h, and 11i, where each row of figures corresponds to results on the Unstructured Mesh pattern with 0 percent (Figs. 11a, 11b, and 11c), 50 percent (Figs. 11d, 11e, and f), and 100 percent (Figs. 11g, 11h, and 11i) of the process topology randomized. In the 0 percent case, we fully determinize the process topology and observe similar error rates to those in the other communication patterns. We observe as the process topology becomes more randomized, the error rates for the predictions using process ID and logical timestamps increase, while the predictions relying on logical ticks remain stable and in the same range as the other communication patterns (i.e., within the 10 percent error range).

Our empirical observations show that the WLST kernel equipped with logical time and logical tick vertex labels better fit cases where receive/send-side non-determinism is the predominant expression of non-determinism. Our observations also show that the WLST kernel equipped with process ID vertex labels better fit cases where non-determinism in process topology is the predominant expression of non-determinism. We contend that the process of determining the right vertex labeling scheme for a given communication pattern can be performed via an automated search (e.g., by using a surrogate-based modeling [36] approach in which the space of possible combinations of vertex labeling and graph kernels can be efficiently explored). This represents a smaller investment of resources than attempting to quantify the degree of non-determinism by hand.

## 4 Identifying Sources of Non-Determinism

Our framework can automatically identify abrupt spikes in kernel distance. These spikes imply sudden changes in the

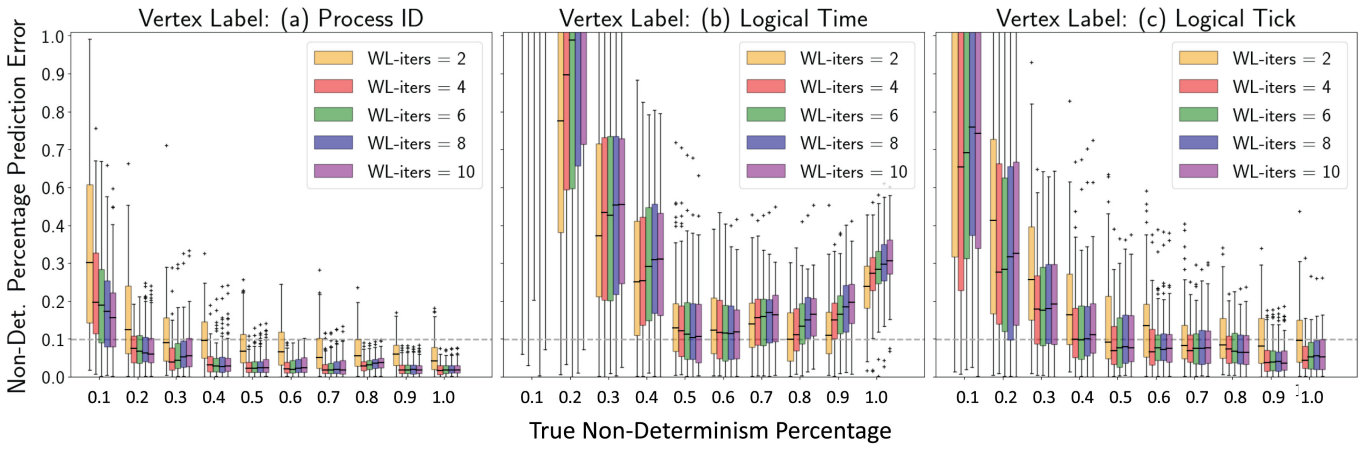



Fig. 10. Relative prediction errors for the non-determinism percentage on the MCB Grid communication pattern.

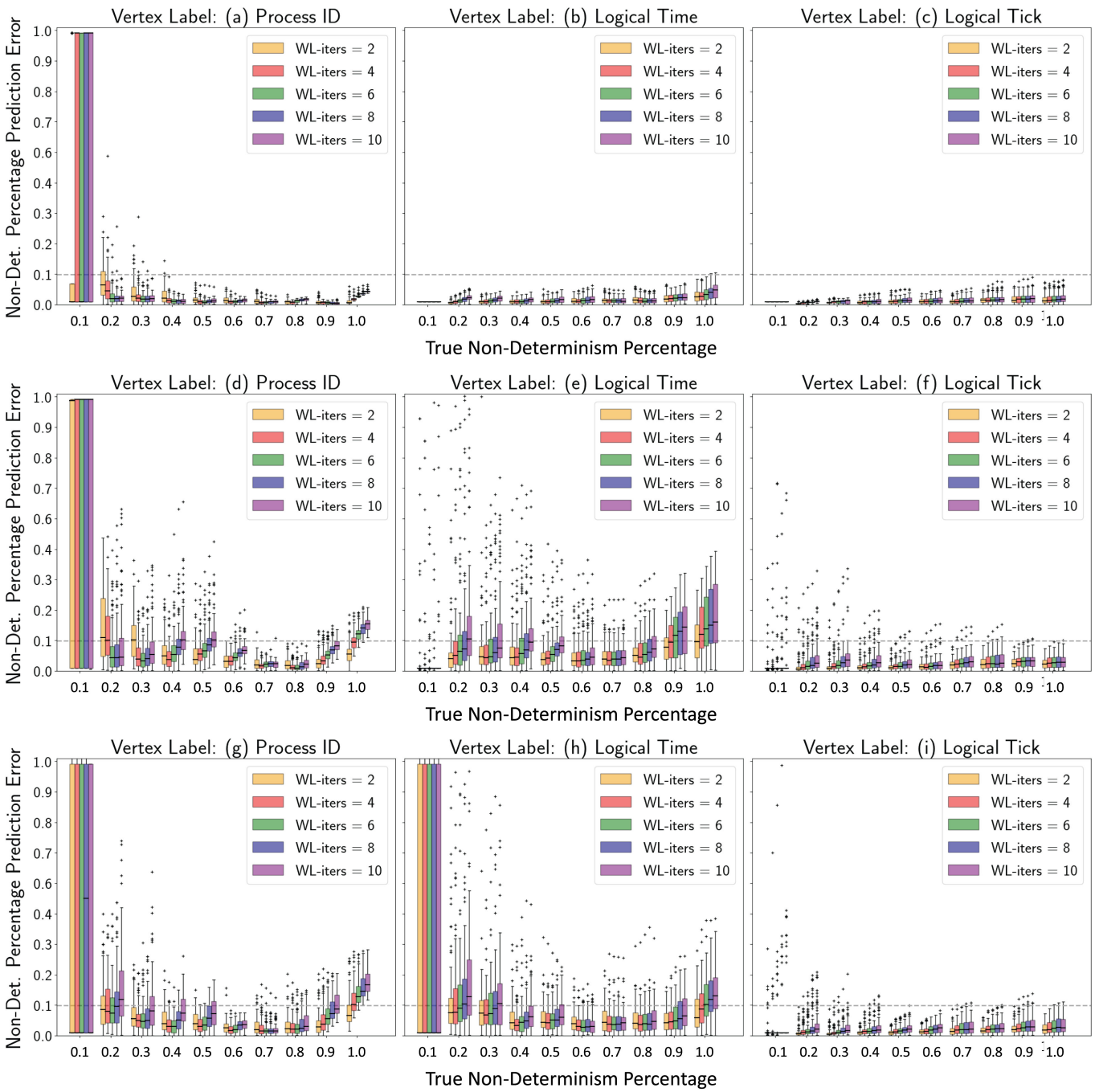



Fig. 11. Relative prediction errors for the non-determinism percentage on the Unstructured Mesh pattern (Process topology non-determinism fractions of 0, 50 percent, and 100 percent are presented in the first, second, and third rows respectively).

underlying communication non-determinism across executions and by-proxy they may be associated to code regions where other aspects of execution state may occur, such as random failures of the execution or sensitive numerical computations. We study two different scenarios in which our framework replaces time-demanding serial examinations of code regions causing non-determinism. The first scenario links the root causes of non-determinism to function calls in adaptive mesh refinement application. The second scenario links the causes to the application's configuration parameters used to generate the executable in a Monte Carlo application. Both applications exhibit non-deterministic behavior.

### 4.1 miniAMR: Non-Determinism From Function Calls

We validate the effectiveness of our framework to identify root causes of non-determinism embedded in prominent chains of function calls by analyzing miniAMR [37]. This is a proxy application for adaptive mesh refinement (AMR) from the Mantevo benchmark suite [38]. We select miniAMR for this proof-of-concept study because it has the communication patterns of real AMR codes, such as Enzo, the AMR code we refer to in our example in Section 2. By using our framework, we automatically identify the callstacks most related to non-determinism using our automated kernel distance-based workflow. In doing so, we determine

that callstacks related to mesh refinement and load-balancing routines in miniAMR contribute most to non-determinism. We validate our findings by manually inspecting the miniAMR code to confirm that our framework extracts the same knowledge about the non-determinism with no *a priori* knowledge of the source code, or the underlying phenomena it simulates.

#### 4.1.1 Analyzing Kernel Distance Time Series for miniAMR

We first build the temporal partition of the event graphs. We construct event graphs from traces of miniAMR executions. Each execution runs on 64 processes for 100 time steps, performing an additional step of mesh refinement once every five time steps and load-balancing once per mesh refinement. We partition each event graph into a sequence of subgraphs that we refer to as "slices," resulting in a sequence of 120 slices. Each slice is identified automatically by leveraging the two consecutive global synchronizations (e.g., `MPI_Barrier` and blocking collectives) that delimit each time step of miniAMR. Specifically, each pair of global synchronizations defines an interval of logical time. The slice bounded by those synchronizations is the subgraph consisting of all vertices with logical timestamps in that interval. Each slice represents either a single time step of miniAMR, during which a 3D-stencil communication pattern runs, or a mesh refinement event, during which blocks are split into sub-blocks (i.e., mesh refinement) and processes exchange ownership of blocks to maintain load-balance. Mesh refinement occurs every five time steps, thus each slice sequence is made up of 20 repetitions of five stencil communication slices followed by a mesh refinement slice, for a total length of 120 slices.

Then, we create the kernel distance time series. We execute miniAMR 100 times, create the event graph for each execution, and partition the graphs into slices. The slices are indexed from 0 to $n-1$, with the earliest occurring slice given the smallest index. Slices with the same index represent communications at the same time steps across the different executions. Our workflow capitalizes on the properties of production applications that are becoming increasingly prevalent on HPC systems (i.e., ensembles and uncertainty quantification pipelines) [39], [40]. Such applications are executed in large numbers of runs, thus the data our framework needs to identify root causes of non-determinism can be collected by default during testing and development. We compare the slices with the same index to measure the communication non-determinism within specific time steps. We thus obtain a sequence of sets of kernel distances that we refer to as the *kernel distance time series* (KDTS). For computing the KDTS, we use the WLST kernel with logical timestamp vertex labels and 40 WLST iterations for all kernel distance computations. In Fig. 12, we provide a graphical description of how the KDTS is obtained from the initial set of miniAMR event graphs.

Last, we identify anomalous slices. In the KDTS, extreme kernel distance values indicate significant communication non-determinism, and abrupt changes in kernel distance indicate shifts in the underlying communication patterns. We want to identify the slices in which these shifts occur

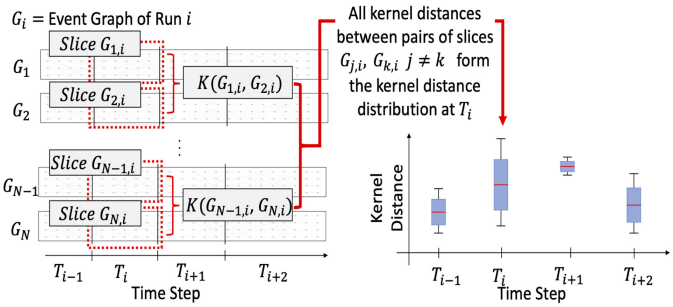



Fig. 12. Kernel distances between time-delimited "slices" of miniAMR event graphs form a time series of kernel distance distributions.

automatically. To this end, we employ two-sample Kolmogorov-Smirnov tests to determine whether the kernel distances from one slice are likely sampled from the same underlying distribution as those of its immediate predecessor and its immediate successor. The null hypothesis of the tests is that its two samples originate from the same distribution. We flag a slice as "anomalous" if both tests (i.e., the test against its predecessor slice and against its successor slice) reject the null hypothesis. Once this test has been performed for all slices, we extract the callstack labels from each anomalous slice. We compute the percentage with which different callstacks occur. If certain callstacks occur with higher percentage in the anomalous slices as compared to non-anomalous slices, then we posit that these routines contribute most to the non-determinism.

#### 4.1.2 Framework Effectiveness on miniAMR Runs

In Fig. 13, we show the KDTS extracted from miniAMR event graphs over 120 slices, where 100 of the slices represent stencil communication steps, and the remaining 20 represent mesh refinement events. For each slice, we plot the set of kernel distances (blue box plots with y-axes on the left). The alternation between periods of stencil communication (i.e., the groups of five low-mean, low-variance boxes) and mesh refinement events (i.e., the periodic high-mean, high-variance boxes) clearly illustrates that kernel distance peaks during mesh refinement. This indicates that miniAMR's mesh refinement and load-balancing communication patterns are significantly more non-deterministic than its stencil communication pattern. In the same figure we plot the volume of blocks (i.e., individual pieces of miniAMR's mesh decomposition) whose owning process changes during each mesh refinement step (i.e., the red line with y-axes on the right). We observe that the kernel distances during mesh refinement correlate strongly with the volume of exchanged blocks, suggesting that the more blocks change owners, the more drastically the communication pattern changes, thus leading to greater kernel distances. We compute Pearson's $R$ and Spearman's $\rho$ correlation scores between kernel distance and block traffic of 0.7 ($p = 0.0006$) and 0.68 ($p = 0.0008$) respectively. This further indicates that mesh refinement and load-balancing are the driving forces behind miniAMR's non-determinism.

Fig. 14 shows a comparison between the categories of callstacks observed in anomalous slices (a) and those observed

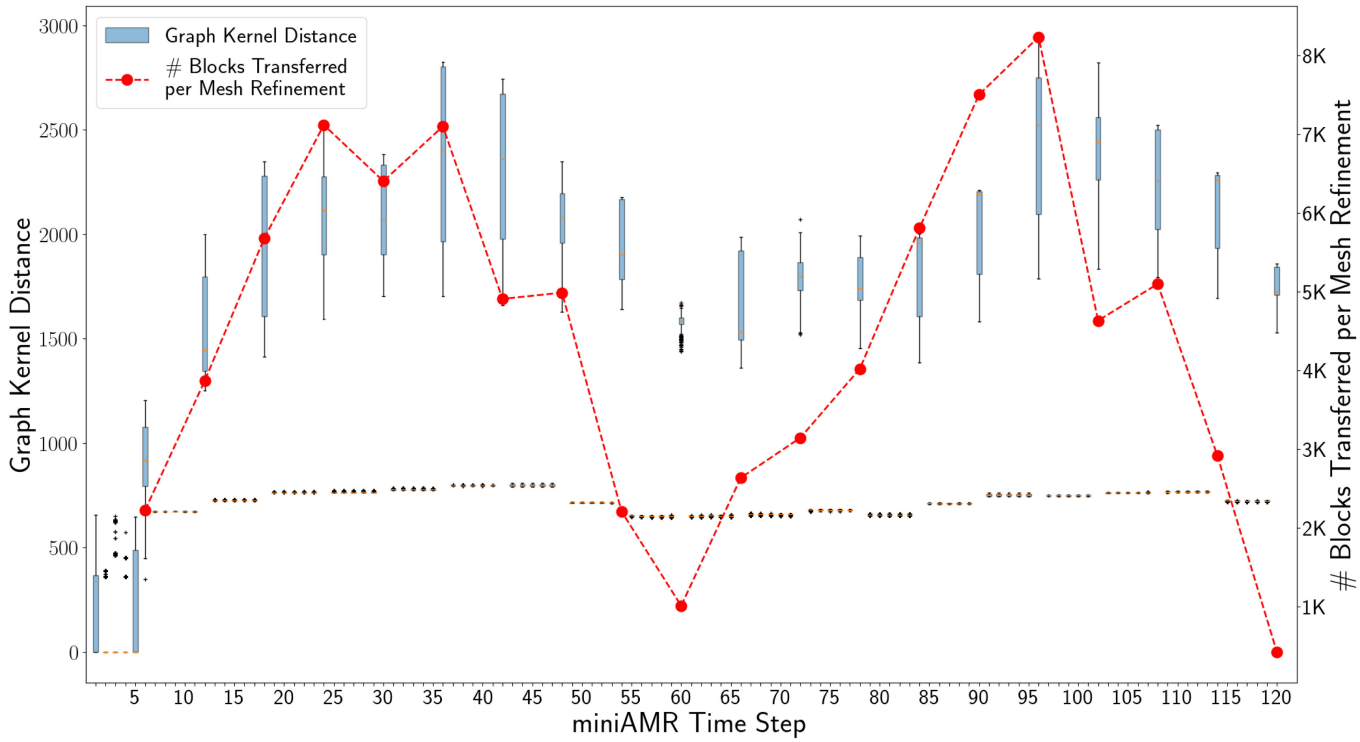



Fig. 13. miniAMR time step versus graph kernel distance distribution (blue box plots with y-axes on the left) and volume of block traffic (the red line with y-axes on the right).

in non-anomalous slices (b). Callstacks involved in mesh refinement and load-balancing cumulatively make up $\approx 90\%$ of all callstacks extracted from anomalous slices. In contrast, mesh refinement and load-balancing callstacks comprise a vanishing minority of the total in the non-anomalous slices. This result demonstrates that our workflow can locate the root causes of non-determinism purely by analyzing changes over time in kernel distances between executions.

### 4.1.3 Validation Against Manual Analysis

The utility of our framework for identifying root causes of potentially harmful non-determinism is proportional to how closely its results match with those obtained through an exhaustive manual analysis of the source code. In Fig. 14, we show that our framework identifies call paths related to mesh refinement and load-balancing as principally responsible for the most extreme and anomalous periods of non-determinism over the course of our sampled miniAMR executions. To validate that these results are meaningful and useful, we conduct a manual analysis of the miniAMR source code to identify which call paths in fact contribute to non-determinism.

Our manual analysis of miniAMR's source code consists of identifying the call paths terminating in MPI functions that have inherently non-deterministic semantics, e.g., `MPI_Testsome`. We initially search for MPI features that can induce receiver-side non-determinism, which we enumerate in Table 4. Our initial search identifies `MPI_Waitany` as the only MPI feature that *directly* contributes to non-determinism, we analyze the miniAMR callgraph to determine all of the unique callpaths that terminate in `MPI_Waitany`, as shown in Fig. 15. We immediately observe that the vast majority of these callpaths are rooted at the `refine` function, indicating that miniAMR's mesh refinement routines contribute significantly to its non-determinism. This comports with the expected behavior of AMR applications in general, namely that as mesh elements are refined, the underlying communication pattern changes in a dynamic and unpredictable way, resulting in the necessary use of non-deterministic communication constructs. This is

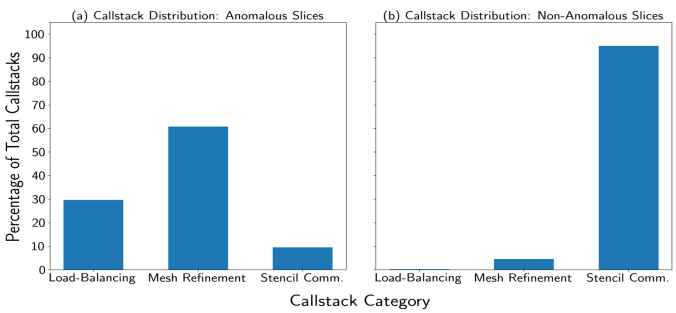



Fig. 14. Distributions of callstacks by category from (a) anomalous slices and (b) non-anomalous slices.

TABLE 4
MPI Features Capable of Inducing Receiver-Side Non-Determinism

| Feature | Examples |
|---|---|
| Wildcard Parameters | MPI_ANY_SOURCE |
| Wildcard Matching Functions | MPI_Waitany, MPI_Testany |
| Multi-Request Matching Functions | MPI_Waitsome, MPI_Testsome |
| Non-Blocking Probes | MPI_Iprobe |

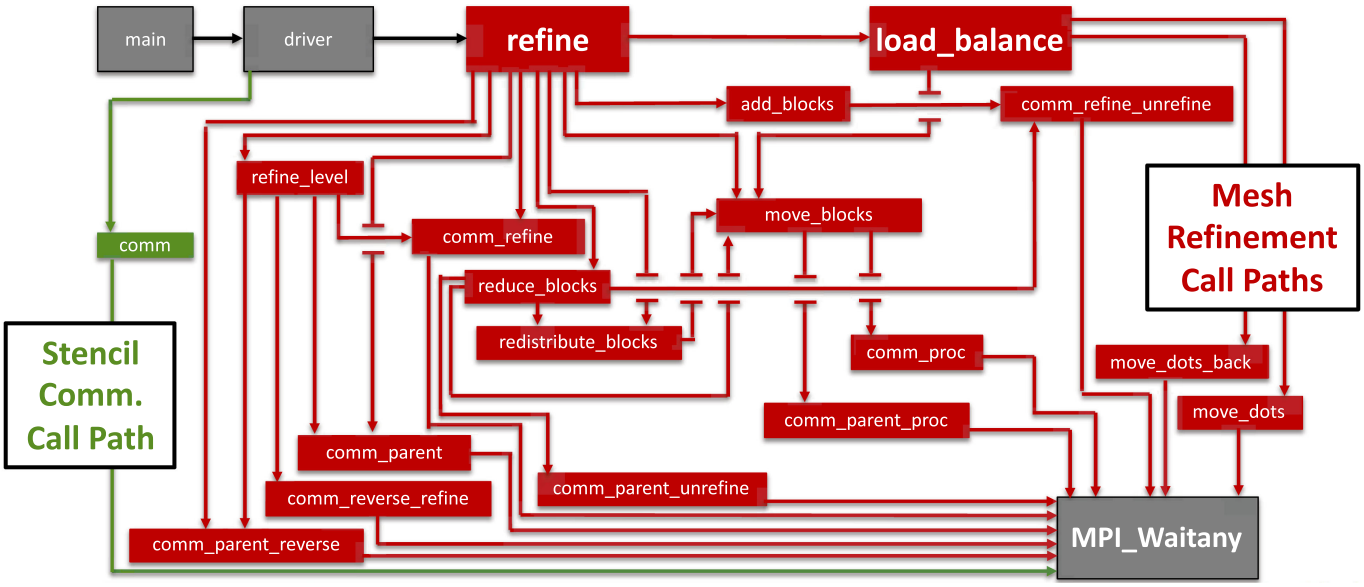
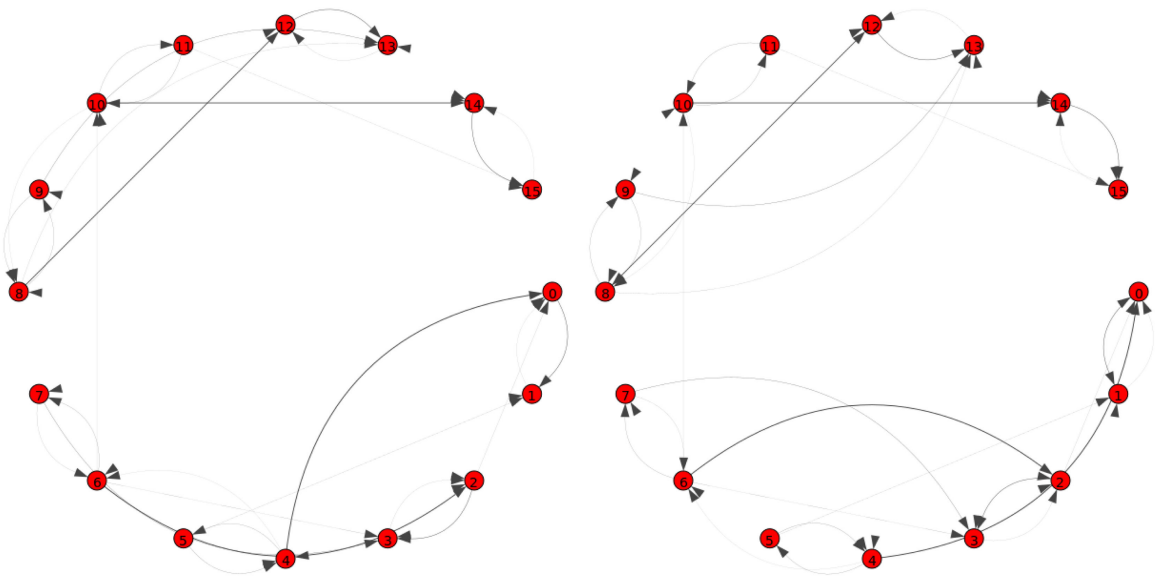



Fig. 15. Call paths in miniAMR terminating in `MPI_Waitany`. Red indicates call path implements mesh refinement or load-balancing. Green indicates call path implements stencil communication.

amplified by the changes in communication patterns induced by miniAMR's load-balancing, as can be seen in the subset of `MPI_Waitany` callpaths rooted at `refine` → `load_balance`.

While the prevalence of `MPI_Waitany` calls in its mesh-refinement and load-balancing routines indicates that miniAMR exhibits receiver-side non-determinism, it is less straightforward to determine whether it exhibits sender-side or process topology non-determinism simply by examining its source code. To address these concerns, we instrumented miniAMR to log a record of how mesh blocks (i.e., pieces of the domain decomposition that can be split during mesh refinement and change owner during subsequent load-balancing) are transferred during mesh refinement events.

In Fig. 16, we show the exchange of mesh blocks between processes during the same mesh refinement event for two separate 16-process runs. In these graphs, vertices represent MPI processes and a directed edge $i \rightarrow j$ represents a transfer of at least one block from $p_i$ to $p_j$. Edge thickness denotes the number of blocks transferred, with thicker edges representing a relatively greater number of blocks. Thus, we can directly observe that receiver-side, sender-side, and process-topology non-determinism exists in miniAMR's mesh refinement and load-balancing communication pattern.



(a) (b)

Fig. 16. Comparison of miniAMR block traffic during a mesh refinement event for two runs (a) and (b). Vertices represent MPI processes, directed edges represent a transfer of block ownership, and edge thickness represents the number of blocks transferred. Observe that in run (a), $p_4$ transferred blocks to $p_0$ during this mesh refinement event, but the same transfer did not occur in run (b). Similarly, in run (b), $p_6$ transferred blocks to $p_2$ during this mesh refinement event, but the same transfer did not occur in run (a).

### 4.2 MCB: Non-Determinism Arising From Configuration Parameters

We validate the effectiveness of our framework to identify root causes of non-determinism embedded in application's configuration parameters (i.e., values defined at compile time or at the beginning of execution that control application behavior) by analyzing Monte Carlo Benchmark (MCB) [33] runs. MCB is a proxy application for Monte Carlo simulations based on three main types of particle exchange patterns: a particle exchange on a 2-dimensional grid, a non-blocking gather of particles, and a non-blocking scatter of particles. The latter two serve as bookkeeping for the particle exchange. Note that the MCB Grid communication patterns used in Section 3.3 was extracted from this code; here we consider the code in its entirety. MCB is implemented primarily using MPI's non-blocking point-to-point communication primitives, specifically making heavy use of non-blocking matching functions such as `MPI_Testsome` during any particle exchange.

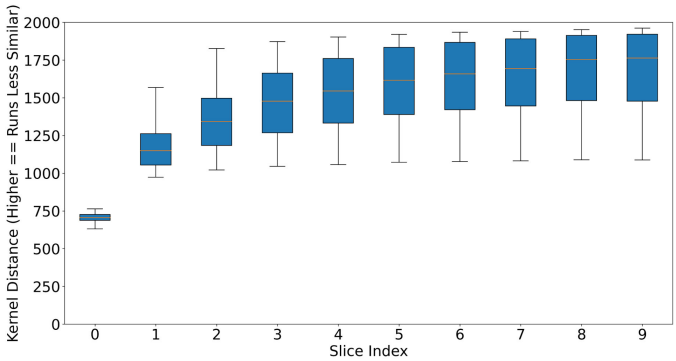
Our framework reveals changes in the levels of non-determinism in communication patterns due to the size of the buffer used to store exchanged particles. Specifically, the framework allows us to measure the sensitivity of the three specific particle exchange patterns to different buffer size configurations. A communication buffer is shared between each distinct pair of neighbor processes. By varying the size of the communication buffer one can vary the rate of communication (e.g., by decreasing the buffer size we increase the number of messages issued). Often the MCB user is not aware of the buffer size used during the compilation and is left to guess whether the observed numerical non-determinism is linked to the unknown buffer size.

By using our framework, we automatically quantify how much changes in buffer size induce changes in the level of non-determinism manifested in MCB communication patterns (i.e., particle exchange grid, non-blocking gather, and non-blocking scatter) as driven by variability of the non-deterministic point-to-point communication events these communication patterns are composed of. Non-determinism in communication patterns may result in differences in numerical outputs from run to run [35]. Our framework allows us to peel apart executions and glean the relationship between the application's configuration parameters and the non-determinism in the individual communication patterns.

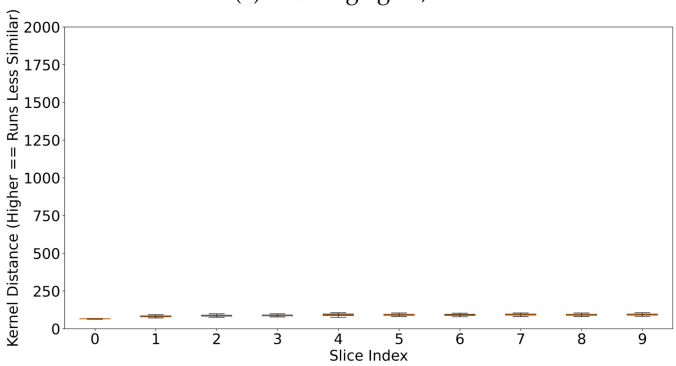
#### 4.2.1 *Framework Effectiveness on MCB Runs*

We compare the degree of non-determinism in terms of graph kernel distances for the MCB applications when using 1M particles per process and a buffer size of either 5 or for 5,000 (i.e., a buffer containing 5 to 5,000 particles in transit between different pairs of neighbor processes).

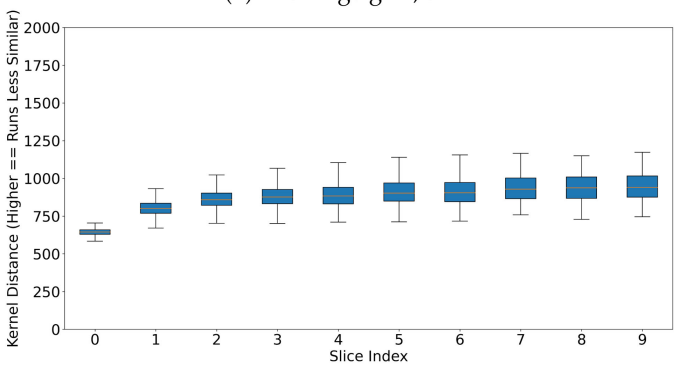
In Fig. 17 we summarize the kernel distances of MCB for two of the three MCB particle exchange patterns (i.e., particle exchange on a 2D grid in Figs. 17a and 17b, and non-blocking gather in Figs. 17c and 17d). The third MCB particle exchange pattern is not shown as its kernel distances are close to zero, indicating that the pattern does not exhibit significant non-determinism. We run ten MCB steps (or framework slices) for each simulation, with two distinct MCB executables with buffer size 5 and buffer size 5K respectively. For each buffer size, we record 100 executions of MCB with our framework. We conduct our tests on Vulcan,



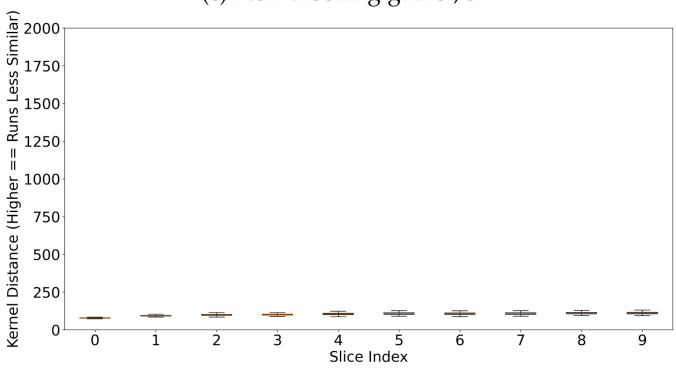

(a) Exchange grid, 5




(b) Exchange grid, 5K




(c) Non-blocking gather, 5




(d) Non-blocking gather, 5K

Fig. 17. Kernel distance time series for MCB's particle exchange grid communication pattern and non-blocking gather communication pattern for a MCB simulation of 1M particles per process with buffer size 5 in (a) and (c) and buffer size 5K in (b) and (d).

a BlueGene/Q cluster at Lawrence Livermore National Laboratory. Each node of Vulcan consists of 16 1.6 GHz PowerA2 processors and is equipped with 16 GB of RAM. The nodes are networked to each other in a 5D torus.

The figures outline how the combination of many particles being simulated while using a small buffer (in Figs. 17a and 17c) induces significantly more non-deterministic communication than the other configuration with larger buffer size (in Figs. 17b and 17d). By forcing more frequent messaging (i.e., by reducing the buffer size), we are fundamentally increasing the amount of communication taking place and thereby increasing the degree to which randomness in message timing can contribute to run-to-run differences in message order. Our framework automatically quantifies the level of non-determinism for each of MCB's communication patterns, providing a guide to diagnosing the root causes of possible difference in numerical scientific outcome due to the level of observed non-determinism as measured by kernel distances.

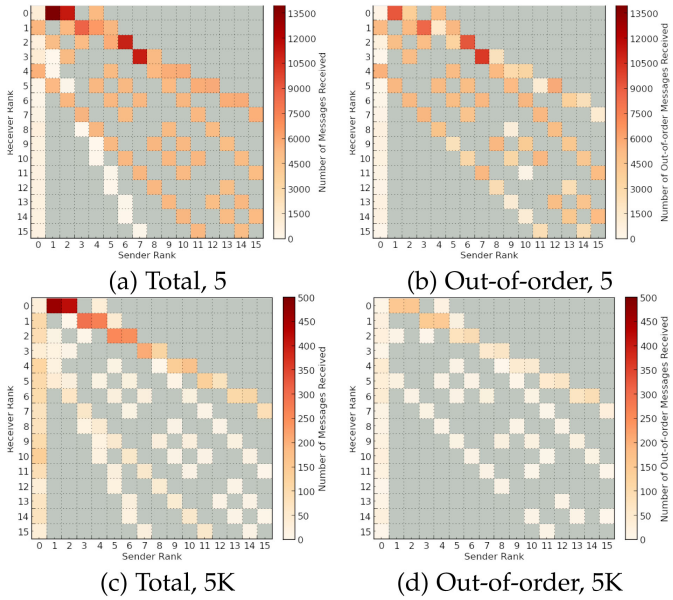



Fig. 18. Total number of messages (a and c) and total number of out-of-order messages (b and d) received by each receiving process $i$ per sender process $j$ for a parameter configuration of 1M particles per process using buffer size 5 and 5K.

### 4.2.2 Validation Against Record & Replay Tools

The utility of our framework for quantifying impacts of an application's configuration parameters on non-determinism is assessed by capitalizing on the features of a state-of-the-art Record & Replay (R&R) tool, ReMPI [35]. Note that ReMPI was initially designed to deterministically replay executions exhibiting rare bugs and not for capturing root sources of non-determinism. In [41], we showed how intermediary data collected by the tool, such as out-of-order MPI messages, can be used as a proxy for quantifying non-determinism. ReMPI uses the Lamport clocks to distinguish between MPI messages that are received in an expected order versus a non-expected order. Empirically, the more non-deterministic a communication pattern, the more messages are received "out-of-order". The mechanism by which these two notions (level of non-determinism and out-of-order messages) are linked is as follows. During a recording phase (application's execution), the MPI function calls are collected by ReMPI in a record. The ReMPI record format is a data structure that contains sufficient information about interprocess communication events to enable deterministic replay of the recorded execution. Specifically, the record uses permutation encoding as described in [35] in order to store to memory only out-of-order messages for the replay of the application with the same order of message exchanges, making them a proxy for the cost of recording as well as the level of non-determinism in executions. In other words, the rate of out-of-order MPI messages can serve as a proxy for non-determinism through their memory usage: the larger the memory, the more non-deterministic the executions.

Experimental results with ReMPI can indicate which of an application's configuration parameters (e.g., MCB's buffer size) has a measurable effect on the number of out-of-order messages received by all processes over the course of executions and consequently on the non-determinism. To this end, we use ReMPI to generate the heatmaps of total and out-of-order messages sent and received by processes during the three MCB communication patterns.

Figs. 18a and 18c show the total number of messages received by each receiving process from each process that sent to it for executions. This data is collected using the same platform used to collect the results presented in Fig. 17 (i.e., on the 16-process Vulcan system at LLNL); results were recorded using ReMPI with MPI_SEND-ticking and are the average over 100 runs.

Figs. 18a and 18b refer to high communication intensity (i.e., 1M particles per process, with a buffer size of 5); and Figs. 18c and 18d refers to low communication intensity (i.e., 1M particles per process, with a buffer size of 5K).

The heatmaps in Figs. 18 outline how some processes only receive from some other processes during certain patterns. For example, process $P_0$ does not receive messages from process $P_2$ during the neighbor-to-neighbor particle exchange, but does receive messages from process $P_2$ during the non-blocking gather. The heatmaps in Figs. 18a and 18b also confirm the three known particle exchange patterns: (1) on the upper triangular matrix of a heatmap, the non-blocking gather of particles; (2) along the diagonal, the particle exchange on a 2-dimensional grid; and (3) on the lower triangular matrix, the non-blocking scatter of particles.

Figs. 18b and 18d show heatmaps counting only the out-of-order receives. This data is a subset of that shown in Figs. 18a and 18c respectably, and refers to the same three communication intensity and floating-point workload scenarios studied above. The inspection of the heatmaps of out-of-order messages outlines which one, if any, of the three communication patterns impacts non-determinism (i.e., the non-blocking gather pattern in the upper upper matrix and the particle exchange on a 2-dimensional grid along the diagonal). Similarly to the outcome with our framework, we observe that cells indicating the greatest number of out-of-order receives correspond to messages sent during the non-blocking gather communication pattern and particle exchange on a 2-dimensional grid, whereas the non-blocking scatter pattern exhibits none. The intensity of the heatmap cells indicate attribution of out-of-order receives, and by proxy non-determinism, for the different

TABLE 5
Median Runtime Overheads (% of base application runtime) Imposed by DUMPI versus Our P$^N$MPI-Based Tracing Stack (DUMPI + CSMPI) on Executions of miniAMR and MCB

| Application | Tracing Configuration | Number of MPI Processes | | | |
|---|---|---|---|---|---|
| | | 64 | 512 | 1024 | 2048 |
| miniAMR | DUMPI | 0.72 | 1.99 | 2.29 | 4.37 |
| | DUMPI + CSMPI | 2.39 | 5.83 | 6.73 | 8.48 |
| MCB | DUMPI | 0.56 | 5.17 | 7.23 | 10.6 |
| | DUMPI + CSMPI | 1.70 | 5.33 | 7.38 | 11.1 |

configuration parameters. ReMPI results match the observations generated by our framework. While reaching similar conclusions as our framework, ReMPI results were obtained after ad-hoc manually extractions of embedded information from intermediary data, a much more time-demanding and less scientist-friendly operation.

### 4.3 Framework Overheads

To assess the overheads of our framework, we first measure the runtime overhead required by our framework to trace the executions. We compare the baseline overhead of DUMPI versus our P$^N$MPI-based tracing stack (both DUMPI and CSMPI) on runs of miniAMR and MCB at scales up to 2,048 processes. Table 5 shows that at 2,048 processes, our tracing stack imposes less than 10 percent median overhead in terms of application run time. This overhead compares favorably with the overheads imposed by similar tools used in production environments, such as record-and-replay tools [35], indicating that our workflow would be an efficient debugging and reproducibility assessment aid for HPC applications.

Next, we measure the space overhead required to trace the executions of miniAMR and MCB that we analyze above. For the 100 executions of miniAMR we trace, the median total size of DUMPI trace files per run is 0.7 GB and the median total size of CSMPI trace files per run is 0.16 GB. Thus, the median total size of trace files for our tracing stack is 0.86 GB per run (i.e., our tracing stack imposes $\sim$23% more space overhead than DUMPI alone in the median case). Given that the CSMPI traces provide the critical link between runtime non-determinism and root causes in source code and the fact that CSMPI traces represent a fraction of the total trace size compared to the DUMPI traces, we deem this additional overhead acceptable.

## 5 Related Work

In recent years, increasing awareness of the need to manage non-determinism in parallel applications [2] has given rise to several software solutions. The two solutions that most closely resemble our framework are PopMine [6] and SABALAN [5]. PopMine [6] is a tool that analyzes traces of message orders in MPI applications, also represented by an event graph-like structure to determine a minimal DAG that triggers a given bug. In contrast with PopMine, our framework does not search for a message order that triggers a specific bug. Our framework enables a generalized notion of anomaly detection for non-deterministic communication patterns and thus enables localization of non-determinism that impacts scientific correctness (e.g., through the interaction between non-deterministic communication and floating-point non-associativity) rather than being restricted to bugs that cause crashes or hangs.

SABALAN [5] is a tool that analyzes trace data, albeit not from MPI applications, and finds hierarchies of motifs that can potentially be linked in bug manifestation. In contrast with SABALAN, our framework compares communication patterns across multiple executions, rather than searching for possibly non-deterministic execution motifs within a single run.

In production HPC environments, record-and-replay tools [41] currently allow users to record a non-deterministic application's execution and then replay it exactly, thus enabling reproducibility of non-deterministic bugs. State of the art record-and-replay tools such as ReMPI [35] target production-scale runs and prioritize scalability, both in terms of runtime and the record size. Other record-and-replay tools target hybrid MPI+OpenMP executions [42], MPI applications using one-sided communication [43], [44], replay of isolated subgroups of processes [45], and probabilistic replay [46]. In addition, tools such as NINJA [1] are used in conjunction with record-and-replay tools to improve the chances of capturing non-deterministic bugs. In contrast with record-and-replay tools, our framework focuses on localizing non-determinism and mapping it back to previously unknown root causes, rather than determinizing executions.

## 6 Conclusion

In this paper we present a trace analysis framework for characterizing non-determinism in HPC applications. We develop a novel enrichment of existing event graph models specifically designed to link runtime manifestations of non-determinism to their root causes. We demonstrate the viability of graph kernel similarity as a proxy for the degree of non-determinism across a range of communication patterns. Finally, we evaluate our end-to-end workflow on two use cases: miniAMR, demonstrating that our framework can recover knowledge about the root causes of non-determinism in an automated fashion; and MCB, demonstrating quantifying non-determinism arising from changes in configuration parameters. Our workflow has the potential to enable substantial savings in human effort during the development and debugging of complex, non-deterministic scientific applications. Our future work targets adaptively selecting effective graph kernels for different communication patterns and transitioning our framework to operate at runtime as a dynamic analysis.

## Acknowledgments

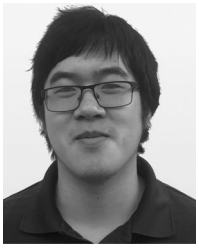
This work was supported in part by the National Science Foundation under Grant 1900888 and Grant 1900765.

## References

[1] K. Sato, D. H. Ahn, I. Laguna, G. L. Lee, M. Schulz, and C. M. Chambreau, "Noise injection techniques to expose subtle and unintended message races," in *Proc. 22nd ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2017, pp. 89–101.

[2] G. Gopalakrishnan *et al.*, "Report of the HPC correctness summit, Jan 25–26, 2017, Washington, DC," 2017, *arXiv:1705.07478*.
[3] V. Stodden and M. S. Krafczyk, "Assessing reproducibility: An astrophysical example of computational uncertainty in the HPC context," in *Proc. 1st Workshop Reproducible, Customizable Portable Workflows HPC*, 2018, pp. 1–5
[4] D. Chapp, T. Johnston, and M. Taufer, "On the need for reproducible numerical accuracy through intelligent runtime selection of reduction algorithms at the extreme scale," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2015, pp. 166–175.
[5] S. Alimadadi, A. Mesbah, and K. Pattabiraman, "Inferring hierarchical motifs from execution traces," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng.*, 2018, pp. 776–787.
[6] E. Seo, M. M. H. Khan, P. Mohapatra, J. Han, and T. F. Abdelzaher, "Exposing complex bug-triggering conditions in distributed systems via graph mining," in *Proc. Int. Conf. Parallel Process.*, 2011, pp. 186–195.
[7] F. Cappello, A. Guermouche, and M. Snir, "On communication determinism in parallel HPC applications," in*Proc. 19th Int. Conf. Comput. Commun. Netw.*, 2010, pp. 1–8.
[8] I. Laguna, R. Marshall, K. Mohror, M. Ruefenacht, A. Skjellum, and N. Sultana, "A large-scale study of MPI usage in open-source hpc applications," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2019, pp. 1–14.
[9] R. D. Falgout and U. M. Yang, "HYPRE: A library of high performance preconditioners," in *Proc. Int. Conf. Comput. Sci.*, 2002, pp. 632–641.
[10] R. M. Ferencz, "Technical spotlight: NEAMS structural mechanics with Diablo," Lawrence Livermore Nat. Lab., Livermore, CA, USA, *Tech. Rep. TR-LLNL-TR-645482*, 2013.
[11] P. Langlois, R. Nheili, and C. Denis, "Recovering numerical reproducibility in hydrodynamic simulations," in *Proc. IEEE 23rd Symp. Comput. Arith.*, 2016, pp. 63–70.
[12] M. Taufer, O. Padron, P. Saponaro, and S. Patel, "Improving numerical reproducibility and stability in large-scale numerical simulations on GPUs," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2010, pp. 1–9.
[13] M. Baranowski *et al.*, "Reproducing ParConnect for SC16," *Parallel Comput.*, vol. 70, pp. 18–21, 2017.
[14] G. Williams *et al.*, "SC16 student cluster competition challenge: Investigating the reproducibility of results for the ParConnect application," *Parallel Comput.*, vol. 70, pp. 27–34, 2017.
[15] K. Sato *et al.*, "Pruners: Providing reproducibility for uncovering non-deterministic errors in runs on supercomputers," *Int. J. High Perform. Comput. Appl.*, vol. 33, no. 5, pp. 777–783, 2019.
[16] G. Sawaya, M. Bentley, I. Briggs, G. Gopalakrishnan, and D. H. Ahn, "FLiT: Cross-platform floating-point result-consistency tester and workload," in *Proc. IEEE Int. Symp. Workload Characterization*, 2017, pp. 229–238.
[17] G. L. Bryan *et al.*, "Enzo: An adaptive mesh refinement code for astrophysics," *Astrophys. J. Suppl. Series*, vol. 211, no. 2, 2014, Art. no. 19.
[18] J. Wilke, "Structural simulation toolkit (SST) DUMPI trace library." Accessed: Mar. 31, 2020. [Online]. Available: https://github.com/sstsimulator/sst-dumpi
[19] A. F. Rodrigues *et al.*, "The structural simulation toolkit," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 4, pp. 37–42, 2011.
[20] J. Gait, "A probe effect in concurrent programs," *Softw. Pract. Experience*, vol. 16, no. 3, pp. 225–233, 1986.
[21] M. Schulz and B. R. De Supinski , "A flexible and dynamic infrastructure for MPI tool interoperability," in *Proc. Int. Conf. Parallel Process.*, 2006, pp. 193–202.
[22] M. Schulz and B. R. De Supinski , "PnMPI tools: A whole lot greater than the sum of their parts," in *Proc. ACM/IEEE Conf. Supercomput.*, 2007, pp. 1–10.
[23] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.
[24] D. Kranzlmüller, Event graph analysis for debugging massively parallel programs, 2000, Ph.D. dissertation, Dept. for Graphics and Parallel Process. John Kepler Univ. Linz, Linz, Austria.
[25] D. Kranzlmüller, S. Grabner, and J. Volkert, "Event graph visualization for debugging large applications," in *Proc. SIGMETRICS Symp. Parallel Distrib. Tools*, 1996, pp. 108–117.
[26] S. V. N. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt, "Graph kernels," *J. Mach. Learn. Res.*, vol. 11, no. 40 pp. 1201–1242, 2010.
[27] N. Shervashidze, P. Schweitzer, E. J. V. Leeuwen, K. Mehlhorn, and K. M. Borgwardt, "Weisfeiler-Lehman graph kernels," *J. Mach. Learn. Res.*, vol. 12, pp. 2539–2561, 2011.
[28] J. M. Phillips and S. Venkatasubramanian, "A gentle introduction to the kernel distance," 2011, *arXiv:1103.1625*.
[29] B. Weisfeiler and A. A. Lehman, "A reduction of a graph to a canonical form and an algebra arising during this reduction," *Nauchno-Technicheskaya Informatsia*, vol. 2, no. 9, pp. 12–16, 1968.
[30] S. Ghosh, N. Das, T. Gonçalves, P. Quaresma, and M. Kundu, "The journey of graph kernels through two decades," *Comput. Sci. Rev.*, vol. 27, pp. 88–111, 2018.
[31] U. Kang, H. Tong, and J. Sun, "Fast random walk graph kernel," in *Proc. SIAM Int. Conf. Data Mining*, 2012, pp. 828–838.
[32] J. Park, M. Smelyanskiy, U. M. Yang, D. Mudigere, and P. Dubey, "High-performance algebraic multigrid solver optimized for multi-core based distributed parallel systems," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2015, pp. 1–12.
[33] N. Gentile and B. Miller, "Monte carlo benchmark (MCB)," Lawrence Livermore Nat. Lab., Livermore, CA, USA, *Tech. Rep. TR-LLNL-CODE-507091*, 2010.
[34] N. Jain and A. Bhatele, "Chatterbug communication proxy applications suite," Lawrence Livermore Nat. Lab., Livermore, CA, USA, *Tech. Rep. LLNL-CODE-756471*, 2018.
[35] K. Sato, D. H. Ahn, I. Laguna, G. L. Lee, and M. Schulz, "Clock delta compression for scalable order-replay of non-deterministic parallel applications," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2015, pp. 1–12.
[36] T. Johnston, M. Alsulmi, P. Cicotti, and M. Taufer, "Performance tuning of MapReduce jobs using surrogate-based modeling," *Procedia Comput. Sci.*, vol. 51. pp. 49–59. 2015. [Online]. Available: https://doi.org/10.1016/j.procs.2015.05.193
[37] C. T. Vaughan and R. F. Barrett, "Enabling tractable exploration of the performance of adaptive mesh refinement," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2015, pp. 746–752.
[38] M. A. Heroux *et al.*, "Improving performance via mini-applications," Sandia Nat. Lab., Albuquerque, NM, USA, *Tech. Rep. SAND2009–5574*, 2009.
[39] T. Dahlgren *et al.*, "Scaling uncertainty quantification studies to millions of jobs," in *Proc. 27th ACM/IEEE Int. Conf. High Perf. Comput. Commun. Conf. (SC)*, 2015, pp. 1–2.
[40] S. Herbein *et al.*, "Scalable I/O-aware job scheduling for burst buffer enabled HPC clusters," in *Proc. 25th ACM Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2016, pp. 69–80.
[41] D. Chapp, K. Sato, D. Ahn, and M. Taufer, "Record-and-replay techniques for HPC systems: A survey," *Supercomput. Front. Innov.*, vol. 5, no. 1, pp. 11–30, 2018.
[42] S. Budanur, F. Mueller, and T. Gamblin, "Memory trace compression and replay for SPMD systems using extended PRSDs," *Comput. J.*, vol. 55, no. 2, pp. 206–217, Feb. 2012.
[43] X. Qian, K. Sen, P. Hargrove, and C. Iancu, "SReplay: Deterministic sub-group replay for one-sided communication," in *Proc. Int. Conf. Supercomput.*, 2016, pp. 17:1–17:13.
[44] X. Qian, K. Sen, P. Hargrove, and C. Iancu, "OPR: Deterministic group replay for one-sided communication," in *Proc. 21st ACM SIGPLAN Symp. Princ. Prac. Parallel Program*, 2016, pp. 47:1–47:2.
[45] R. Xue, X. Liu, M. Wu, Z. Guo, W. Chen, W. Zheng, Z. Zhang, and G. Voelker, "MPIWiz: Subgroup reproducible replay of MPI applications," in *Proc. 14th ACM SIGPLAN Symp. Princ. Prac. Parallel Program.*, 2009, pp. 251–260.
[46] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu, "PRES: Probabilistic replay with execution sketching on multiprocessors," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Princ.*, 2009, pp. 177–192.

**Dylan Chapp** received the PhD degree in computer science from the University of Delaware in 2020. He is currently a researcher associate with the Global Computing Lab, Department of Electrical Engineering and Computer Science, The University of Tennessee, Knoxville. His research interests include characterization of nondeterminism in parallel applications and reproducibility in high-performance computing.



**Nigel Tan** received the BS degree in both computer science and applied math from the University of California Merced and the MS degree in computational and applied math from Rice University. He is currently working toward the PhD degree in computer science with The University of Tennessee, Knoxville. He was an intern with Lawrence Berkeley National Lab exploring HPX performance on the Cori supercomputer and Los Alamos National Lab porting the Vector-Particle-In-Cell (VPIC) project using Kokkos. His research interests include high-performance computing with an emphasis on performance portability and optimization across multiple architectures.

**Sanjukta Bhowmick** received the PhD degree in computer science from Pennsylvania State University, University Park in 2004. She is currently an associate professor Computer Science and Engineering Department, University of North Texas. Her research interests include the study of the properties of large, dynamic complex networks and using high-performance computing to analyze them, understanding how noise affects network analysis, and developing uncertainty quantification for network analysis.

**Michela Taufer** (Senior Member, IEEE) received the PhD degree in computer science from the Swiss Federal Institute of Technology (ETH) in 2002. She is currently an ACM distinguished scientist and the Jack Dongarra professor of high-performance computing with the Department of Electrical Engineering and Computer Science, The University of Tennessee, Knoxville. Her research interests include high-performance computing, volunteer computing, scientific applications, scheduling and reproducibility challenges, and in situ data analytics.