Locating and Categorizing Inefficient Communication Patterns in HPC Systems Using Inter-Process Communication Traces

Luay Alawneh[a,*], Abdelwahab Hamou-Lhadj[b]

[a] *Faculty of Computer and Information Technology*
Jordan University of Science & Technology
Irbid, Jordan
[b] *Department of Electrical & Computer Engineering*
Gina Cody School of Engineering and Computer Science
Concordia University
Montreal, Quebec, Canada

Abstract

High Performance Computing (HPC) systems are used in a variety of industrial and research sectors to solve complex problems that require powerful computing platforms. For these systems to remain reliable, we should be able to debug and analyze their behavior in order to detect root causes of potential poor performance. Execution traces hold important information regarding the events and interactions among communicating processes, which are essential for the debugging of inter-process communication. Traces, however, tend to be considerably large, hindering their applicability. In previous work, we presented an approach for automatically detecting communication patterns and segmenting large HPC traces into execution phases. The goal is to reduce the effort of analyzing traces by allowing software analysts to focus on smaller parts of interest. In this paper, we propose an approach for detecting and localizing inefficient communication patterns using statistical and trace segmentation methods. In addition, we use the Analytic Hierarchy Process to categorize slow communication patterns based on their severity and complexity levels. Using our approach, an analyst can quickly locate slow communication patterns that may be the cause of important performance problems. We show the effectiveness of our approach by applying it to large traces from three HPC systems.

Keywords: Message Passing Interface, Distributed systems, Performance analysis, Debugging, Software tracing, Data Analytics

1. Introduction

The demand for High Performance Computing (HPC) systems continues to grow to meet the needs of many industrial and research sectors such as bioinformatics, medical information processing, and financial analytics, for powerful systems to process and solve large and complex problems [1]. The popularity of HPC programs has further flourished with the advent of multicore and cloud computing environments.

HPC programs that are developed using the Message Passing Interface (MPI) standard [2] rely on a large number of processes working together by exchanging messages to solve computationally intensive problems. MPI combines processes in different groups called *Communicators*. Processes in one *communicator* interact with each other according to a virtual topology, which usually follows a linear, 2 or 3-dimensional mesh structure. Processes communicate with their nearest or non-nearest neighbors in the mesh. In a typical MPI program, these communications are repetitive and form communication patterns. A communication pattern groups sequences of MPI communication events from different processes that are working towards a specific task. The binary tree and butterfly patterns are examples of communication patterns [3]. Figure 1 shows the butterfly and the binary tree patterns for eight processes.

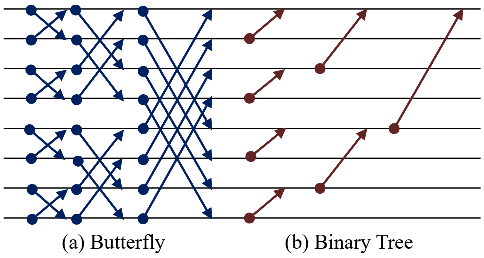

(a) Butterfly (b) Binary Tree

Figure 1: Communication Pattern Examples

Performance analysis and debugging of HPC systems require dynamic analysis techniques due to the distributed nature of these systems. An early work presented by Preissl et al. [4] showed that automatic identification of communication patterns from execution traces can be useful for understanding an application's communication behavior, that would eventually facilitate debugging and performance analysis tasks. The problem is that typical traces can be overwhelmingly large with many instances of various communication patterns. The mere detection of communication patterns may still generate a lot of data that is hard for a software analyst to grasp. To alleviate this problem, researchers have proposed the concept of trace segmentation in

[*] Corresponding author
Email addresses: lmalawneh@just.edu.jo (Luay Alawneh), wahab.hamou-lhadj@concordia.ca (Abdelwahab Hamou-Lhadj)

which a large trace is partitioned into distinct segments, which depict execution phases of the traced scenario, and detect communication patterns within each segment [5][6][7].

An execution phase is broadly defined as a region within the trace that contains communication patterns, which implement a specific program functionality [7]. Trace segmentation is also used to support program comprehension tasks in monolithic systems [8]. The main objective is to provide a way for the software analyst to only focus on parts of the trace of interest instead of browsing the whole trace. Casas et al. [5] and Chetsa et al. [9] showed how MPI trace execution phases can help with performance optimization tasks by uncovering regions in a trace with the highest latency. Isaacs et al. [6] presented a trace visualization and analysis tool that logically orders and visualizes the MPI communication behavior into fine-grained phases to determine the lateness in program operations using temporal metrics and visual inspection.

In our previous work [7], we proposed an effective trace segmentation approach, which involves two main steps. In the first step, we detect communication patterns in the entire trace using natural language processing techniques. In the second step, we use the extracted communication patterns to identify dense homogeneous clusters, which represent distinct execution phases of the trace. This is achieved using information theory concepts such as Shannon entropy [10] and the Jensen-Shannon Divergence measure [11]. The new contributions of this paper are summarized as follows:

- We improve our previous technique for segmenting traces into execution phases using the Akaike Information Criterion (AIC) [12] to identify finer execution phases.
- We extend the communication pattern detection approach by using distinctive events to identify the boundaries of coherent communication events in each process trace, which facilitates the detection of process repeating patterns.
- We propose an approach for detecting inefficient communication pattern instances in trace segments using statistical analysis. More specifically, we use the Median Absolute Deviation (MAD) and the Modified Z-score measures [13] to determine slow communication patterns.
- We propose an approach for the categorization of communication patterns using the Analytic Hierarchy Process (AHP) [14] by examining the complexity and severity levels for slow patterns in execution phases.
- We demonstrate the effectiveness of our approach by applying it to five large traces generated from three different HPC systems.
- Through the analysis of a sample of inefficient patterns detected by our approach, we provide a detailed discussion on the potential root causes, which demonstrate the usefulness of our approach in practice.

The rest of the paper is organized as follows. Section 2 presents a background of HPC and sequence segmentation followed by related studies on techniques for the analysis of MPI programs in Section 3. Section 4 details the proposed approach. In Section 5, we apply our approach on several traces generated from HPC systems and show how it could detect patterns of inefficient behavior. We conclude our paper in Section 6 and discuss future directions.

2. Background

This section starts by providing a more detailed view of HPC and communication patterns. Then, it presents the sequence segmentation technique that we use in our study for identifying computational phases.

2.1. A More Detailed View of HPC & Communication Patterns

Complex industrial and scientific applications rely on HPC systems to process data in a reasonable time. HPC systems use shared and distributed memory programming paradigms. Processes in a distributed memory environment collaborate using message passing, defined based on the MPI standard. The MPI specification provides point-to-point, collective, and one-sided communications. It also supports blocking and non-blocking modes of communication. The point-to-point operations contain the message tag, communicator, and size of exchanged data attributes. An example of an MPI operation is `MPI_Waitall`, which makes the process wait for a group of non-blocking events to complete.

MPI programs follow the Single Program Multiple Data (SPMD) and the Multiple Program Multiple Data (MPMD) programming models [15]. In SPMD, MPI decomposes the data into different processes that use the same program, whereas, in MPMD, processes execute different programs with different data. The behavior of MPI systems takes the form of communication patterns, which are composed of either point-to-point or collective communication events.

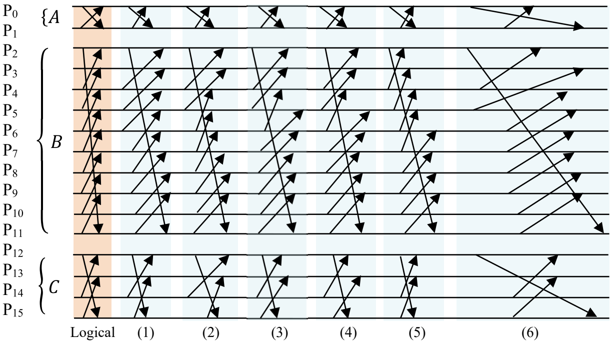
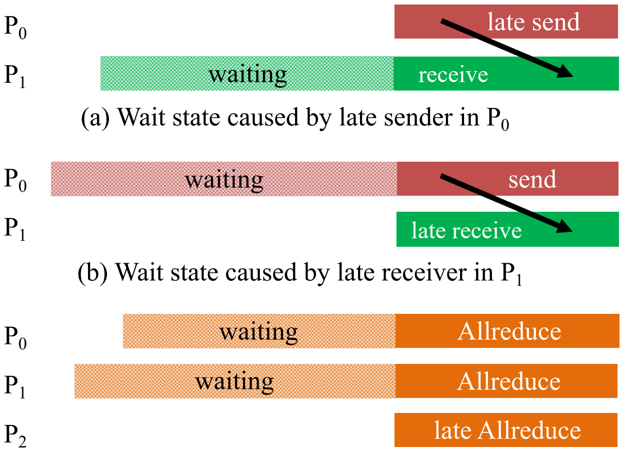


Figure 2: Sample MPI Trace

Figure 2 shows 16 processes ($P_0 - P_{15}$) exchanging point-to-point messages. A communication pattern contains only the processes that are involved in message exchange. Therefore, we consider A, B, and C as distinct patterns. These patterns are repeating 6 times in the trace. Patterns A, B, and C involve processes $P_0 - P_1$, $P_2 - P_{11}$, and $P_{12} - P_{15}$ respectively. The *logical* column represents the logical ordering of the events without any delays while columns $(1-6)$ depict the real communication with different durations.

At a high-level of abstraction, a typical MPI program consists of three main phases, which are the *Initialize*, *Setup*, and *Solve* phases. These three phases contain sub-phases based on different functionalities. In each phase, the processes communicate based on various patterns. Identifying these phases helps in localizing the analysis of the program, and provides fine-grained and coherent views of the trace. For example, let us consider that we have six execution phases in the trace in Figure 2, where each phase corresponds to one column. Consequently, each phase contains 3 different communication patterns where the communications in Phase 6 are slower than the previous instances.

In message passing programs, the latency is characterized as a wait state where a process remains idle until it synchronizes with the delayed counterparts. Figure 3 shows three different types of latency. Figure 3a shows that the receiving process P_1 enters a wait state due to the late-sender P_0. Figure 3b shows that process P_0 enters a wait state since process P_1 is late in posting its receive operation. Figure 3c shows that processes P_0 and P_1 entered a wait state due to the slow P_2 process during collective communication. These kinds of delays occur due to several reasons such as load imbalance, network congestion, longer transmission routes, resource limitations, and large and excessive message transfers. These delays may cause the latency to propagate across the communicating processes, resulting in communication imbalance.



(a) Wait state caused by late sender in P$_0$

(b) Wait state caused by late receiver in P$_1$

(c) Wait state caused by late collective in P$_2$

Figure 3: Latency Examples in Message Passing Programs

2.2. *Sequence Segmentation*

In this section, we present a sequence segmentation technique that we use in our approach to segment the trace into more coherent phases. The algorithm relies on Shannon entropy [10] to measure the level of randomness in the sequence. Low entropy suggests that the sequence is rather homogeneous and may not be further segmented. Equation 1 measures Shannon entropy H of sequence S, where N is the length of the sequence, k is the number of distinct symbols, and N_j is the frequency of the j^{th} symbol in the sequence. Moreover, N_j/N is the probability of the j^{th} symbol in the sequence.

$$H = \sum_{j=1}^{k} \frac{N_j}{N} \log \frac{N_j}{N} \qquad (1)$$

To find the segmentation point in the sequence, we need to calculate the entropy for the left (S_l) and the right (S_r) segments at each point in the sequence. Equations 2 and 3 measure the entropy for S_l and S_r at point i in S respectively.

$$H_l = \sum_{j=1}^{k_l} \frac{N_j^l}{i} \log \frac{N_j^l}{i} \qquad (2)$$

$$H_r = \sum_{j=1}^{k_r} \frac{N_j^r}{N-i} \log \frac{N_j^r}{N-i} \qquad (3)$$

To find the segmentation point, we use the Jensen-Shannon Divergence (D_{JS}) [11] to measure the similarity between S_l and S_r at each point in S using Equation 4.

$$D_{JS} = H - \frac{i}{N} H_i - \frac{N-i}{N} H_r \qquad (4)$$

The point with the highest D_{JS} value indicates the segmentation position. To determine when to stop segmenting a sequence, we apply the Bayesian Information Criterion (BIC) model selection criterion [16]. Equation 5 represents the BIC measure for model selection where L is the maximum likelihood of the model, while K represents the number of free parameters in the two models.

$$BIC = -2 \log L + K \log N \qquad (5)$$

BIC is used to check whether the model M_1 (represented by S), or M_2 (represented by its direct subsequences S_l and S_r) fit the data well and should be selected. The number of free parameters K is calculated using $(k_l + k_r + 1 - k)$, where k_l, k_r, and k represent the number of distinct parameters in S_l, S_r and S respectively. To select the model represented by S_l and S_r, the BIC value should be close to zero (i.e. $\Delta BIC < 0$). Thus, we need to calculate the likelihoods L_1 and L_2 for M_1 and M_2 respectively. Equation 6 calculates L_1 for S where p_j is the probability of finding the j^{th} symbol in S.

$$L_1(S) = \prod_{j=1}^{k} p_j \qquad (6)$$

The value of p_j is the probability of the j^{th} symbol to appear in S and is calculated as N_j/N, where N_j is the number of times the j^{th} symbol appears in S while N is the length of S. Consequently, the log-likelihood of M_1 is measured by Equation 7.

$$\log L_1(S) = \sum_{j=1}^{k} N_j \log \frac{N_j}{N} \qquad (7)$$

By referring to Equation 1, the log-likelihood for M_1 (log $L_1(S)$) is calculated using Equation 8.

$$\log L_1(S) = -NH \qquad (8)$$

Similarly, the likelihood for M_2 is measured using Equation 9 where p^l_j is equal to N^l_j/N and p^r_j is equal to N^r_j/N.

$$L_2(S_l, S_r) = \prod_{j=1}^{k_l} p^l_j \prod_{j=1}^{k_r} p^r_j \tag{9}$$

Hence, Equation 10 measures the log-likelihood for M_2.

$$\log L_2(S_l, S_r) = \sum_{j=1}^{k_l} N^l_j \log \frac{N^l_j}{N} + \sum_{j=1}^{k_r} N^r_j \log \frac{N^r_j}{N} \tag{10}$$

By referring to Equation 1, the log-likelihood for M_2 (log $L_2(S_l, S_r)$) is calculated using Equation 11.

$$\log L_2(S_l, S_r) = -N_l H_l - N_r H_r \tag{11}$$

The relative increase of the log-likelihood of the two models is calculated as $\log(L_2/L_1)$ which is equal to $\log(L_2) - \log(L_1)$. Using Equations 8 and 11, the value of $\log(L_2/L_1)$ is calculated using Equation 12.

$$\log \frac{L_2}{L_1} = NH - (N_l H_l + N_r H_r) \tag{12}$$

By referring to Equation 4, the value of ND_{JS} is calculated using Equation 13.

$$ND_{JS} = NH - (N_l H_l + N_r H_r) \tag{13}$$

The maximum likelihood is set at the point with the highest ND_{JS} value. To further segment the sequence, the BIC value should be close to zero (i.e. $\Delta BIC < 0$). By substituting $N\hat{D}_{JS}$ (where $\hat{D}_{JS}$ is the maximum D_{JS} value) for L in Equation 5, we get the inequality shown by Equation 14.

$$2N\hat{D}_{JS} > K \log N \tag{14}$$

Thus, we can segment sequence S if $\hat{D}_{JS}$ is greater than $(K \log N)/2N$. Similarly, we can use the positive segmentation strength s in Equation 18 to determine further possible segmentation of the sequence. The value of s is determined by the relative increase of $2N\hat{D}_{JS}$ from the BIC threshold ($K \log N$) value [17]. Therefore, for $\hat{D}_{JS}$ with a positive s value, model M_2 will be selected (i.e. S is segmented). Otherwise, sequence S will not be further segmented.

$$s = \frac{2N\hat{D}_{JS} - K \log N}{K \log N} \tag{15}$$

3. Related Work

Several tools have been developed for the visualization of MPI traces to facilitate program comprehension and system analysis tasks [18][19]. Although trace visualization tools capture details regarding the whole execution trace, it is difficult to analyze and comprehend the program execution by mere reliance on visualization techniques. For example, Figure 4 shows a zoomed-in view of a trace of 16 processes using the Vampir [18] visualization tool. This typical view of the trace is cluttered, making it challenging to analyze the communication behavior and find the slow events and patterns without some sort of trace analysis techniques.

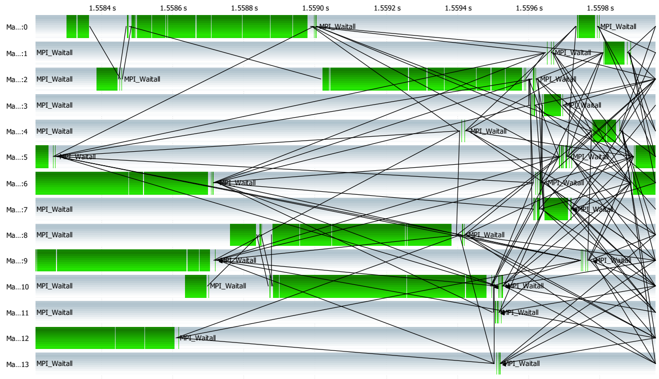


Figure 4: Cluttered view of SMG2000 with 16 Processes

Many approaches and tools [20][19] used dynamic analysis to guide in the performance analysis process. For example, several studies used the program's critical path to determine performance issues [21][22]. However, critical paths may ignore some interesting execution paths during analysis. Wei et al. [23] presented MPI-RCDD for runtime detection of deadlocks by logging MPI communications. The algorithm depends on the message timeout and process dependency mechanisms in identifying deadlocks. In the following, we present related studies of execution trace analysis techniques for program comprehension and performance analysis in MPI programs.

3.1. Online Monitoring Approaches

Mao et al. [24] proposed an approach to identify wait states and quantify them in order to guide in detecting the root cause of latency. They used a lightweight profiler instead of analyzing large execution traces. The proposed approach is scalable and portable with low overhead induced on the running processes. Sikora et al. [25] proposed a dynamic performance abstraction technique to automatically discover causal execution paths (both communication and computational events) in MPI parallel programs to help understand their performance behaviors. This approach helps in online diagnosis of performance issues generated from execution monitoring of the running program. It extracts the application behavior from high level program structures such as loops and communication operations. Then, it labels all program elements with statistical execution profiles. Our approach provides a more fine-grained analysis since it detects all the communication patterns involved in the trace and compares their performance in order to identify inefficient behaviors.

Aguilar et al. [26] used event flow graphs in the context of performance monitoring of MPI applications to keep track of temporal orderings among events and guide in visual performance analysis. They used their profiling approach for runtime identification of loop nesting structures that involve communication events without explicit instrumentation of the source code. They utilize the iterative behavior of MPI parallel programs by only collecting statistical information once the program goes in a stable state, from a small number of iterations, resulting in low overhead on the running program. Usually, the stable state represents the Solve phase since it contains a long iterative loop. The usefulness of this approach is characterized by avoiding the generation of the whole execution trace. In our approach, we need to detect all the communication patterns in the execution trace (containing user-defined and MPI events) and then segment the sequence of detected communication patterns into more homogeneous regions representing the different phases in the program.

Jeannot et al. [27] introduced an introspection monitoring library in Open MPI for runtime optimization of communication time in MPI applications. The main objective is to enable the program to query its state using the monitoring system. The library provides session management, where monitoring can be suspended at any time or can be performed on certain code sections. Moreover, they monitor the collective communications based on their point-to-point implementations which can be used for detecting the affinity among the communicating processes. Ramesh et al. [28] presented an MPI Performance Engineering infrastructure for MPI runtime introspection, online monitoring, and performance tuning recommendations for production and synthetic applications. It is based on the MPI Tools Information Interface (MPI_T) available in the MPI 3.0 standard [2]. It extended the features found in the TAU Performance System, MVAPICH2, and BEACON [19] to fully exploit the features offered by MPI_T.

3.2. *Offline Performance Analysis and Debugging Approaches*

Taheri et al. [29] presented DiffTrace, a performance debugging tool that utilizes the full program trace to differentiate between normal execution trace and a fault-laden trace. The tool provides features for filtering out function calls and presents information about loop-level behavioral differences. They used incremental algorithms to represent loops as concept lattices. Then, they apply hierarchical clustering on these behaviors and determine the inefficient ones.

Gallardo et al. [30] presented MPI Advisor, a tool for providing performance recommendations by tuning MPI configuration parameters. The main purpose of MPI Advisor is the characterization of the main communication behavior of MPI applications and providing recommendations on how to tweak runtime performance. This tool targets application developers who may have insufficient knowledge of the MPI architecture and design. MPI Advisor starts by collecting the data, analyzing the traces, and then providing recommendations on which point-to-point protocols to use, algorithms for implementing collective communications, mapping of MPI tasks to cores, and the Infiniband transport protocol.

Kenny et al. [31] studied the effect of network and its parameters on the performance of MPI programs. They classify the time spent in MPI operations as communication, synchronization, and MPI stack components. They used the Bayesian inference to parameterize the synchronization latency and the MPI stack. By replaying the trace and applying the Bayesian inference, they concluded that the overhead incurred by communication synchronization and the software stack components are significant to the overall performance.

3.2.1. *Analysis of wait states approaches*

Mohr et al. presented KOJAK [32] to locate patterns of inefficient behavior in each process based on wait states in MPI programs. They classify the patterns based on communication durations to determine the ones that are causing the program's inefficient behavior. Scalasca [33], the new version of KOJAK, is a trace analysis tool that is used in the localization of wait states and their root causes as well as the identification of the critical path. The latter is used to measure the feasibility of optimization. Scalasca identifies performance issues such as late message arrivals, and directs the engineer to the source code in order to investigate the root cause of the problem. Each identified performance issue is then assigned a severity level in order to determine whether it should be investigated or not. This approach is limited to identifying the late arrived messages but does not provide information about the inter-process communication behavior.

Bohme et al. [34] proposed an approach for detecting long wait states in message passing programs. They show whether the cause of the delay is from late senders or late receivers, and the subroutines they occur in. However, our approach detects the latency in the context of communication patterns, to enable the understanding of the relations among the communicating processes. Further, we show whether the delay in the pattern is caused by late senders or late receivers, and provide the subroutine that the communication occurs in.

3.2.2. *Performance forecasting using trace extrapolation*

Xing et al. [35] presented ScalaExtrap, a trace-based communication extrapolation for SPMD programs. ScalaExtrap generates a trace for a large number of nodes from a set of traces from a smaller number of nodes. The authors propose to use the extrapolated trace for different purposes such as the assessment of communication requirements by replaying the trace, analysis of inefficient communications, and the evaluation of the program scalability. Tsuji et al. [36] presented SCAlable Mpi Profiler (SCAMP), a tool for extrapolation of MPI traces for scaling up the simulation of larger systems based on existing traces for trace-driven network simulation. SCAMP generates a pseudo MPI event trace from a small-scale execution trace, that fits larger traces from large-scale runs, to be used in network simulation. The pseudo MPI-event trace generator is built using LLVM's intermediate representations. Having an extrapolated version of the current system helps in forecasting the performance behavior of larger instances of MPI applications on the network.

Similarly, Miwa et al. [37] presented Predcom, an approach for approximate prediction of larger scale of communications based on a small scale collected traces. They combined LLVM on source code to capture the program static structure with prediction parameters for the program dynamic behavior. An advantage of their approach is that they can reduce the overhead of generating execution traces without compromising their accuracy. Thus, traces will be generated much faster when compared to capturing all the execution traces at larger scales.

3.2.3. Approaches based on patterns in traces

HPC programs tend to follow specific communication behaviors. Several research studies targeted communication patterns detection in HPC traces as they are helpful in the understanding, debugging, and analysis of these types of systems. Preissl et al. [4] used compressed suffix trees supported by static analysis techniques to guide in detecting communication patterns in MPI traces. The authors suggested using the patterns for performance identification purposes. Our communication pattern detection approach relies on dynamic analysis techniques and does not require building a static model. Knüpfer et al. [38] used the compressed complete call graph (cCCG) to detect the contiguous repeats in HPC traces. This approach is used to achieve an improved trace compression scheme on each process trace but does not consider the detection of communication patterns. Detecting contiguous repeats could also help in identifying the recurring events in each process trace which could be used in the detection of communication patterns.

Trahay et al. [39] proposed a debugging approach to identify the points of interest based on trace summarization. They applied a variation of the LZW compression to detect patterns in each process trace separately. This approach detects the repeating behavior on each process trace and does not consider the detection of communication patterns in the trace. Kunz and Seuren [40] applied finite state automata methods in their communication pattern matching approach. Their approach looks for all the occurrences of an input communication pattern in the whole trace. First, they extract the longest process pattern from the input communication pattern and locate all of its instances in the trace. Then, they construct the communication pattern from each instance (longest pattern) by matching its events with the partner events on their counterparts. This approach is limited to detecting known communication patterns in the HPC trace.

Köckerbauer et al. [41] proposed a debugging approach for message passing programs using pattern matching techniques. The input communication pattern is translated into abstract syntax trees (ASTs) which are then scaled up to the number of processes in the trace. An advantage of this approach is its ability to detect exact and similar communication patterns which can help in the validation of the application behavior. However, it is still limited to known communication patterns. In our approach, we detect the communication patterns in the trace without prior knowledge about the program's communication behavior.

Isaacs et al. [6] used a visualization-based performance analysis approach to guide in the analysis of MPI programs. They use the happened-before relationships of MPI events to extract the execution trace's logical structure. Visualization techniques are then used to cluster related processes based on the program's communication patterns. The logically ordered trace could then be used to locate delayed peer events among the processes. Our approach uses pattern detection, statistical analysis, and information theory techniques in order to identify the program's structure and performance behavior rather than relying on visual inspection.

3.2.4. Approaches based on trace segmentation

Trace segmentation techniques divide the trace into coherent segments that are then mapped to execution phases in the program. A number of studies targeted the identification of computational phases in MPI programs. Gonzalez et al. [42] applied the DBSCAN algorithm, a density-based clustering technique, to detect computational phases in SPMD MPI applications by grouping different CPU bursts collected from performance hardware counters. A computational region is represented by the CPU burst occurring between two consecutive communications. Then, they use defined thresholds to fine-tune the resulting phases. These threshold values could be subjective which may result in inaccurately identified phases. To overcome the shortcomings of the DBSCAN algorithm, the authors improved the quality of the resulting phases by applying Aggregative Cluster Refinement techniques [43]. Chetsa et al. [9] presented the use of execution vectors (hardware performance counters, network communications, disk I/O values) in identifying phases in MPI programs where a new phase is detected when the Manhattan distance between successive vectors is over a specified threshold.

Casas et al. [44] used signal processing techniques to aid in the identification of the main phases in the traces of MPI applications. This approach relies on the iterative behavior of MPI programs in identifying the different computational phases. More precisely, the frequency of the iterative behavior determines the different computational phases in the trace. The Solve phase is labeled as the phase containing most of the parallel iterations. In [5], Casas et al. used CPU computing bursts in conjunction with communication events (signals) in order to define several metrics that could be used in the identification of sub-phases in the Solve (computational) phase. Our approach distinctly identifies the different phases in the program using information theory principles.

3.3. Summary of Related Studies

Table 1 provides a summary of the related studies. The columns show the program analysis and comprehension techniques that they apply, and the features provided by their approaches such as parameter tuning and tool support. Further, we include our proposed approach at the end of the table to position it among the related work. In our approach, we present several trace abstraction techniques to facilitate the understanding of the inter-process communication behavior through communication patterns. We localize these patterns into several phases, where the user can specify the level of granularity of the phases. Further, we study the latency of communication patterns, and categorize the slow ones in each phase based on their severity and complexity levels to guide in selecting the points of interest during analysis.

Table 1: Summary of Related Studies

Study	Techniques	Static Analysis	Profiling	Trace Analysis	Process Repeats Detection	Comm. Patterns Detection	Comm. Patterns Matching	Phase Detection	Latency Identification	Pattern Categorization	Program Comprehension	Performance Optimization	Performance Forecasting	Trace Extrapolation	Online Monitoring	Parameter Tuning	Tool Support
Mao et al. [24]	Score-P call-path profiler: wait-state profiling		●						●								●
Sikora et al. [25]	Performance abstraction	●	●	●					●		●				●		
Aguilar et al. [26]	Loop-nesting structure from event flow graphs		●								●						
Jeannot et al. [27]	Runtime introspection		●									●			●		●
Ramesh et al. [28]	Runtime introspection		●									●			●	●	● ●
Taheri et al. [29]	Jaccard similarity matrices			●					●							●	●
Gallardo et al. [30]	MPI_T and MPI Advisor		●									●				●	●
Kenny et al. [31]	Trace replay, architecture simulation, inference			●					●		●						
Geimer et al. [33]	Call-path metrics		●	●					●		●	●					●
Böhme et al. [34]	Call-path profile, replay of event traces		●	●					●		●	●					●
Xing et al. [35]	Analytical processing, math. transformations			●							●		●	●			
Tsuji et al. [36]	SCAMP			●									●	●			
Miwa et al. [37]	LLVM and parameter prediction	●		●									●	●			
Preissl et al. [4]	Compressed Suffix Trees	●		●	●	●					●						
Knüpfer et al. [38]	compressed complete call graph (cCCG)			●	●						●						
Trahay et al. [39]	LZW compression			●	●						●						●
Kunz and Seuren [40]	Finite state automata			●			●				●						
Köckerbauer et al. [41]	Abstract syntax trees (AST)			●			●				●						●
Isaacs et al. [6]	Logical ordering and Visualization			●					●	●							●
Gonzalez et al. [42]	DBSCAN and Aggregative Cluster Refinement		●					●			●						●
Chesta et al. [9]	Manhattan distance		●					●			●						
Casas et al. [5]	Spectral analysis		●					●			●						
Proposed Approach	Call tree, information theory, MAD, AHP			●	●	●		●	●	●	●					●	●

4. The Approach

Figure 5 shows our overall approach for detecting and locating inefficient communication patterns in MPI traces. We start by extracting the communication patterns by identifying the repeated sequences of MPI calls in each process trace. The output of this step is a sequence of all instances of the detected communication patterns in the trace ordered using the happened-before relationship. Second, we locate the communication patterns within specific trace segments using information theory principles. This helps in reducing the analysis efforts by only focusing on a fine-grained view of the trace. In the third step, we examine the pattern instances to determine the ones that are considerably slower than the other instances using statistical analysis. Finally, the set of inefficient patterns in each trace segment are categorized based on their severity and complexity levels using the AHP technique. In the following, we elaborate on each step by presenting the techniques using examples.

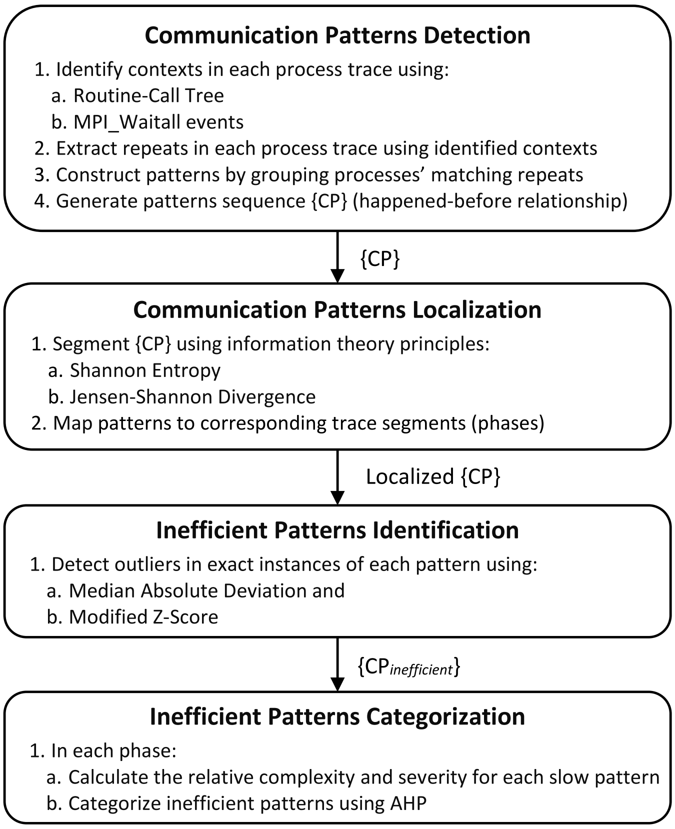
4.1. Detection and Localization of Communication Patterns

The detection of communication patterns consists of two steps. In the first step, we identify all the communication patterns in the trace. In the second step, we segment the trace into execution phases and localize pattern instances in each phase. In the following, we explain the two steps in detail.

4.1.1. Identification of Communication Patterns

The first step of our approach starts by detecting the communication patterns in the trace. To this end, we adapt our pattern detection algorithm presented in [7], which relies on two tasks. First, we iterate through each process trace separately to detect the patterns of MPI communication events. Then, we use these process patterns to construct the final communication patterns.

To illustrate this process, we refer to a sample trace of four processes shown in Figure 6. Figure 6a shows the routine call trees of each process and the MPI calls, where `P` refers to a process, `M` refers to the main function, `F` refers to a user-defined function call, Waitall refers to `MPI_Waitall`, and S and R refer to MPI Send and Receive operations respectively. For example, S4 means Send to process P4, and R1 means receive from process P1. The MPI events in each process trace are extracted and delimited based on the identified contexts as shown in the routine call tree. The contexts are identified based on the user functions they occur in and the `MPI_Waitall` events. Figure 6b



Communication Patterns Detection

1. Identify contexts in each process trace using:
   a. Routine-Call Tree
   b. MPI_Waitall events
2. Extract repeats in each process trace using identified contexts
3. Construct patterns by grouping processes' matching repeats
4. Generate patterns sequence {CP} (happened-before relationship)

Communication Patterns Localization

1. Segment {CP} using information theory principles:
   a. Shannon Entropy
   b. Jensen-Shannon Divergence
2. Map patterns to corresponding trace segments (phases)

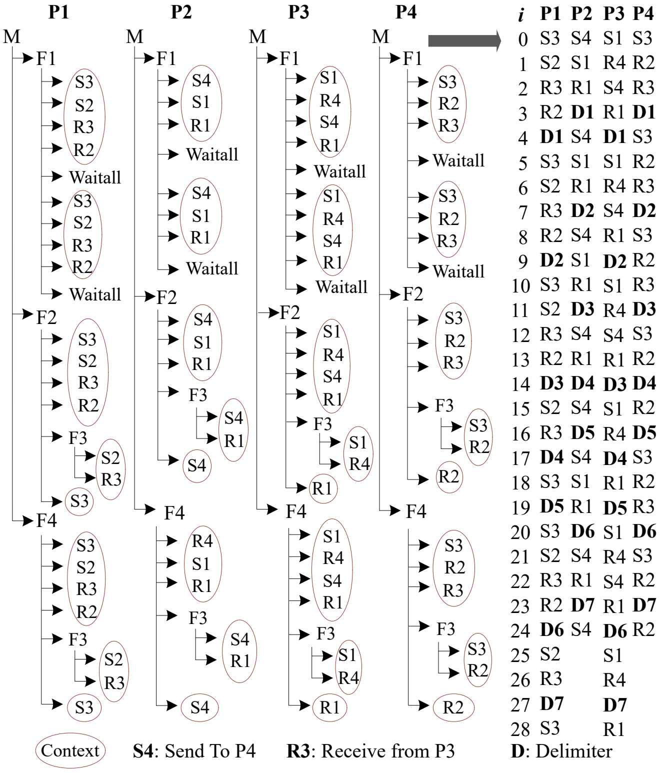
Inefficient Patterns Identification

1. Detect outliers in exact instances of each pattern using:
   a. Median Absolute Deviation and
   b. Modified Z-Score

Inefficient Patterns Categorization

1. In each phase:
   a. Calculate the relative complexity and severity for each slow pattern
   b. Categorize inefficient patterns using AHP

Figure 5: Trace Abstraction Approach For Performance Analysis



i	P1	P2	P3	P4
0	S3	S4	S1	S3
1	S2	S1	R4	R2
2	R3	R1	S4	R3
3	R2	**D1**	R1	**D1**
4	**D1**	S4	**D1**	S3
5	S3	S1	S1	R2
6	S2	R1	R4	R3
7	R3	**D2**	S4	**D2**
8	R2	S4	R1	S3
9	**D2**	S1	**D2**	R2
10	S3	R1	S1	R3
11	S2	**D3**	R4	**D3**
12	R3	S4	S4	S3
13	R2	R1	R1	R2
14	**D3**	**D4**	**D3**	**D4**
15	S2	S4	S1	R2
16	R3	**D5**	R4	**D5**
17	**D4**	S4	**D4**	S3
18	S3	S1	R1	R2
19	**D5**	R1	**D5**	R3
20	S3	**D6**	S1	**D6**
21	S2	S4	R4	S3
22	R3	R1	S4	R2
23	R2	**D7**	R1	**D7**
24	**D6**	S4	**D6**	R2
25	S2		S1	
26	R3		R4	
27	**D7**		**D7**	
28	S3		R1	

D: Delimiter

(a) Routine-Call Trees (b) Delimited Traces

Figure 6: A sample trace with four processes: P_1, P_2, P_3, and P_4

shows the delimited traces that will be used in the extraction of process patterns.

We consider a process pattern as a sequence of MPI events that satisfies two conditions: (1) the sequence cannot be extended to the left and to the right (i.e. maximal repeat [45]), and (2) the sequence must appear within the boundaries of a user function call. For example, in Figure 6, process P_1 has three distinct maximal repeats: S3·S2·R3·R2, S2·R3, and S3.

In our previous algorithm, we used n-gram techniques to detect patterns of MPI events within a user-defined function. In this paper, we use a different approach. We rely on the MPI_Waitall events in conjunction with the user function calls to determine the beginning and ending of a group of MPI events. MPI_Waitall is a common construct in MPI applications where a process waits for a set of MPI communication events to finish a specific task.

Table 2 shows the MPI event patterns (maximal repeats) and their starting positions for each process trace. For example, P_2 contains four instances of pattern PT4 (S4·S1·R1) at positions 0, 4, 8, and 17. It also has two instances of PT5 (S4·R1) at positions 12 and 21. Moreover, it has two instances of the single event pattern PT6 (S4) at positions 15 and 24. The position of the pattern refers to the order of its appearance in the process trace and does not indicate a timestamp.

After detecting the process patterns (repeats), we construct the communication patterns by linking the matching repeats in each process trace. For each communication pattern, the algorithm starts by an empty bag of repeats until all the matching repeats are added. Table 3 shows four distinct communication patterns, their corresponding process patterns, and the number of times they appear in the trace. For example, to construct the first instance of CP2, the algorithm adds PT2 at position 15 to the empty bag. Then, it links the events in PT2 to their matching events in processes P_2 and P_3. Event S2 in PT2 matches event R1 in PT5 at position 12, and event R3 in PT2 matches event S1 in PT8 at position 15. Thus, the instances of PT5 and PT8 will be added to the bag of repeats. Then, event S4 in PT5 is linked with its partner event R2 in PT11 at position 12. Consequently, PT11 will also be added to the bag of repeats. The algorithm will iterate on all the non-visited events in each repeat until all the matching repeats are added to the bag. Hence, the resulting communication pattern consists of PT2, PT5, PT8 and PT11. Once all the instances of communication patterns are constructed, the algorithm assigns a unique symbol to each set of instances that are composed from the same group of repeats.

Finally, we represent the trace as a sequence of communication patterns using temporal ordering (i.e., instances that occur first appear before the others). Table 4 shows the resulting sequence that is used for trace segmentation.

Table 4: Communication Patterns Sequence

1	2	3	4	5	6	7	8	9	10
CP1	CP1	CP1	CP2	CP3	CP4	CP1	CP2	CP3	CP4

Figure 7 shows the detected communication patterns in the sample trace. Communication pattern CP1 contains 14 events (7 messages) while CP2 contains 8 events (4 messages). Further, CP3 and CP4 contain only two events (one message).

Table 2: Maximal Repeats Per Process

P	Process Patterns		Positions in Trace	Frequency
P1	PT1	S3·S2·R3·R2	0, 5, 10, 20	4
	PT2	S2·R3	15, 25	2
	PT3	S3	18, 28	2
P2	PT4	S4·S1·R1	0, 4, 8, 17	4
	PT5	S4·R1	12, 21	2
	PT6	S4	15, 24	2
P3	PT7	S1·R4·S4·R1	0, 5, 10, 20	4
	PT8	S1·R4	15, 25	2
	PT9	R1	18, 28	2
P4	PT10	S3·R2·R3	0, 4, 8, 17	4
	PT11	S3·R2	12, 21	2
	PT12	R2	15, 24	2

Table 3: Communication Patterns Construction

Communication Pattern	Process Patterns	Frequency
CP1	PT1, PT4, PT7, PT10	4
CP2	PT2, PT5, PT8, PT11	2
CP3	PT3, PT9	2
CP4	PT6, PT12	2

4.1.2. Localization of Patterns in Trace Segments

In this step, we partition the sequence of communication patterns into more homogeneous sections. This helps in the localization of patterns, and consequently in reducing the analysis efforts. These cohesive partitions can be viewed as the program computational phases.

This way, software developers can focus only on the phases of interest instead of analyzing the entire trace. To achieve this, we propose an improvement to our trace segmentation algorithm presented in [7], which is inspired by the work of Li et al. [17] on segmenting DNA sequences into more homogeneous regions. The sequence segmentation technique is presented in Section 2.2. In this paper, we apply the Akaike Information Criterion (AIC) [12] for model selection since it tends to penalize complex models less than BIC [46], resulting in more fine-grained phases. Equation 16 represents the AIC measure for model selection where L is the maximum likelihood of the model, while K represents the number of free parameters in the two models.

$$AIC = -2 \log L + 2K \tag{16}$$

Then, similar to BIC, we segment the sequence when the value of AIC is close to zero. By substituting $N\hat{D}_{JS}$ (where $\hat{D}_{JS}$ is the maximum D_{JS} value) for L in Equation 16, we get the inequality shown by Equation 17.

$$2N\hat{D}_{JS} > 2K \tag{17}$$

Finally, the positive segmentation strength s is calculated using Equation 18 to determine further segmentation of the sequence. In the evaluation section, we show a case where BIC fails to further segment the trace.

$$s = \frac{N\hat{D}_{JS} - K}{K} \tag{18}$$

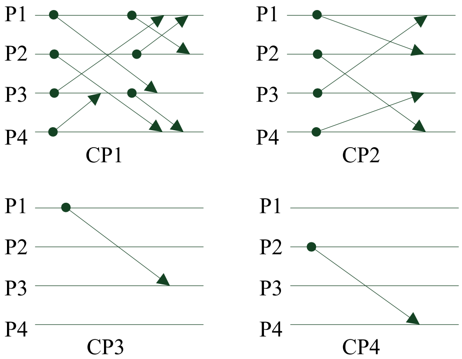


Figure 7: Detected Communication Patterns

The algorithm works on segmenting the sequence of communication patterns recursively. To segment the sequence, we need to find the segmentation point that splits the sequence into more homogeneous sub-segments. Then, the algorithm runs on the left and right sub-segments, and checks whether they can be further segmented. The algorithm will stop until no further segmentation is possible.

The segmentation algorithm results in a binary tree (segmentation tree) where sequence S is the root node at depth zero and the child nodes in the lower levels contain the detected sub-segments. In this paper, we also use the *depth* of the segmentation tree in conjunction with s to terminate the segmentation. Moreover, we use the *length* parameter to prevent segmenting sequences that are shorter than a certain length.

Table 5 shows how the proposed approach segments the list of detected communication patterns S_0, from Section 4.1, into two phases. The maximum D_{JS} value for S_0 is at position 3 with a positive segmentation strength value. The resulting sequences S_1 and S_2 could not be further segmented since the segmentation strength value is negative.

Table 5: Phase Detection for the Trace in Figure 6

i	1	2	3	4	5	6	7	8	9	10
S_0	CP1	CP1	CP1	CP2	CP3	CP4	CP1	CP2	CP3	CP4
d_0	S_0 $s = 1.11$, $\hat{D}_{JS} = 0.42$ at position 3									
d_1	S_1 $s = -1.17$			S_2 $s = -0.08$						

4.2. Identification of Inefficient Communication Pattern Instances

The main objective of this step is to identify the communication pattern instances that take much longer to complete when compared to the other ones. We compare the duration of the instances of the same communication pattern that exchange the same amount of data. The duration of each communication pattern instance is computed as $finish\_time(le) - start\_time(fe)$ where fe and le are the first and last events in the pattern instance, respectively.

We use the modified Z-score method [13] to measure how much the execution time of a given communication pattern instance differs from the typical duration. The advantage of using the modified Z-score over the original Z-score statistic (mean-based measure) is that the latter may be affected by extreme values, which may result in inaccuracies [47]. This is important for MPI programs. In some cases, certain events may take longer time than expected due to network delays or low memory issues, which may be skipped using mean-based methods. The modified Z-score method relies on robust measures, which are the Median and the Median Absolute Deviation (MAD). Equation 19 calculates MAD where $\tilde{x}$ is the sample median.

$$MAD = median(|x - \tilde{x}|) \tag{19}$$

To estimate the standard deviation value for normally distributed data, Equation 19 is multiplied by 1.4826 as shown in Equation 20.

$$\overline{MAD} = 1.4826 \cdot median(|x - \tilde{x}|) \tag{20}$$

Using the estimated MAD value in Equation 20 and the medians, we can calculate the modified Z-score value (M_i) for each element x_i using Equation 21.

$$M_i = \frac{x_i - \tilde{x}}{\overline{MAD}} = 0.6745 \cdot \frac{x_i - \tilde{x}}{MAD} \tag{21}$$

Iglewicz and Hoaglin [13] observed through simulations that a value of $|M_i| > 3.5$ indicates an outlier. In our approach, we will use this value as the default value. However, the tool that implements our approach should allow enough freedom to the analysts to increase this cut-off value based on their judgment.

Table 6: Communication Patterns Information
np: number of processes, *ne*: number of events

Pattern	*np*	*ne*	Time Units Per Instance					
			(1)	(2)	(3)	(4)	(5)	(6)
A	2	4	1	1	2	3	2	8
B	10	18	3	3	4	5	6	12
C	4	6	2	3	3	2	3	9

To explain this method, we use Table 6 to show the duration of the communication patterns found in the trace of Figure 2 in Section 2.1. We suppose that the 1^{st} instance of pattern A takes 1 time unit to complete, while the 6^{th} instance takes 8 time units to complete. Similarly, the 4^{th} instance in pattern B takes 5 time units and the 6^{th} instance of pattern C takes 9 time units.

The modified Z-score determines the pattern instances that deviate from others of the same pattern. Table 7 shows how the M_i value was calculated for each instance. The calculations show that only the last instance (in phase 6) was detected as an outlier.

Table 7: Slow Communication Patterns Identification

i	*Pattern A*			*Pattern B*			*Pattern C*		
	x_i	$x_i - \tilde{x}$	M_i	x_i	$x_i - \tilde{x}$	M_i	x_i	$x_i - \tilde{x}$	M_i
1	1	1	0.6745	3	1	0.6745	2	1	1.3490
2	1	1	0.6745	3	1	0.6745	3	0	0
3	2	0	0	4	0	0	3	0	0
4	3	1	0.6745	5	1	0.6745	2	1	1.3490
5	2	0	0	6	2	1.3490	3	0	0
6	8	6	4.0470	12	8	5.3960	9	6	8.0940
	$\tilde{x} = 2$, $MAD = 1.0$			$\tilde{x} = 4$, $MAD = 1.0$			$\tilde{x} = 3$, $MAD = 0.5$		

After detecting the slow communication patterns, we need to analyze those patterns to determine the cause of the delay. The latency in communication patterns is caused by late senders, late receivers, or a late partner in collective communications. The root cause of these kinds of delays could be related to load imbalance, excessive communications, or network issues. For each slow communication pattern, we provide the *first process to start*, the *last process to start*, the *first process to finish*, and the *last process to finish*. The *first process to start* indicates a waiting process, while the *last process to start* refers to a late process. Further, we present the shortest and the longest events in the pattern.

In a point-to-point communication pattern, we check the event of the *last process to start* to determine whether the latency is caused by a *late-sender* or a *late-receiver*. If it is a send operation, it means that we have a *late-sender*. On the other hand, if the event is a receive operation, it means that the delay is caused by a *late-receiver*.

The analyst can then check the last process to start, in point-to-point and collective communications, and determine whether it was late due to long computations or if it was preceded by excessive communications which caused the process to start its new task late.

4.3. Categorization of Communication Patterns

The final step in our approach is to categorize slow patterns in each trace execution phase. For this purpose, we apply the Analytic Hierarchy Process (AHP) [14]. AHP is a popular multi-criteria decision making and prioritization technique that has been widely used in several problems and domains such as software requirements prioritization [48], finance and banking [49], energy [50], and sustainable development [51].

AHP categorizes alternatives based on multiple criteria. For each criterion, it performs a pairwise comparison between the alternatives to calculate their relative weights. Then, it uses the weighted values from the different criteria to categorize the alternatives in order to help in decision making.

In our context, the different alternatives are the slow communication patterns in each phase. Thus, we need to define the criteria for categorization to help in deciding which patterns to analyze first. A communication pattern that is very slow when compared to the other slow patterns in the same phase requires more attention since it can lead to the root cause of the problem. On the other hand, if the pattern involves many processes and has a large number of events, it will be more difficult to analyze due to its increased complexity. Therefore, we choose the pattern's severity and complexity levels as our criteria for categorization. The pattern severity is related to its duration [52] while the pattern complexity is related to the number of events exchanged among the processes [53][54].

We define three categories for slow communication patterns based on their severity-complexity degree. We refer to this degree as the *Inspection Affinity (IA)* property with *High*, *Medium*, and *Low* levels of affinity. A *High* level means that patterns are extremely severe and less complex. Thus, inspecting patterns in this category should be simpler and may reveal important information regarding latency. Moreover, when a pattern with fewer number of processes and events is extremely slow it means that there should be a clear issue causing the delay since fewer processes should collaborate smoothly. On the other hand, a *Low* level means that the investigation of this pattern may take longer time to investigate due to its increased complexity.

For each criterion, we build a comparison matrix for the patterns in each phase, and then calculate their relative weights. The AHP process will be performed as follows.

1. Estimate the relative weights between patterns in each phase based on their severity.
2. Estimate the relative weights between patterns in each phase based on their complexity.
3. Determine the Inspection Affinity level using the relative weights for each criterion.

In the following, we explain the AHP process using the sample trace in Figure 2 and the corresponding number of processes, events, and duration in Table 6.

4.3.1. *Severity Criterion*

The communication pattern severity is directly related to its duration [52]. Normally, a communication that exchanges more data should take longer to complete. Thus, we determine the pattern severity based on its duration with respect to the size of exchanged data ($duration \div data\_size$). Table 6 shows the duration for each pattern instance. We showed in Section 4.2 that the instances of patterns A, B and C in Phase 6 were detected as outliers using the MAD approach. Assuming that each message holds 1 unit of data. Then, the size of data exchanged in patterns A, B, and C is 2, 9, and 3 respectively. Thus, the corresponding severity values will be 4.0, 1.33, and 3.0 respectively. Table 8a shows the comparison matrix for the three patterns based on their severity level. We normalize the values in each cell by dividing its value by the sum of its corresponding column (value in Σ). Table 8b shows the normalized comparison matrix. The final step is to calculate the weights by averaging the values in each row in the normalized matrix. The *Weight* column in Table 8b shows the relative weights (or the Eigenvector) for the three patterns. This vector will be used in the categorization.

Table 8: Severity-based Weight

	A-6	B-6	C-6
A-6	1.0	4.0/1.33	4.0/3.0
B-6	1.33/4.0	1.0	1.33/3.0
C-6	3.0/4.0	3.0/1.33	1.0
Σ	2.08	6.25	2.78

(a) Comparison Matrix

	A-6	B-6	C-6	***Weight***
A-6	0.48	0.48	0.48	0.48
B-6	0.16	0.16	0.16	0.16
C-6	0.36	0.36	0.36	0.36

(b) Relative Weights (Eigenvector)

4.3.2. *Complexity Criterion*

The complexity of a communication pattern is related to the number of events exchanged among the processes [54]. We calculate the relative pattern complexity as $np \times ne$ where np is the number of processes and ne is the number of events. Table 6 shows that Pattern A involves 2 processes with 4 events, Pattern B involves 10 processes with 18 events, and Pattern C involves 4 processes with 6 events. Thus, the complexity values for patterns A, B, and C will be 8, 180, and 24 respectively. Table 9a shows the comparison matrix for each pair of patterns based on the calculated complexity values. Table 9b shows the normalized values for the cells in Table 9a and the relative weights.

Table 9: Complexity-based Weight

	A-6	B-6	C-6
A-6	1	8/180	8/24
B-6	180/8	1	180/24
C-6	24/8	24/180	1
Σ	26.50	1.18	8.83

(a) Comparison Matrix

	A-6	B-6	C-6	***Weight***
A-6	0.04	0.04	0.04	0.04
B-6	0.85	0.85	0.85	0.85
C-6	0.11	0.11	0.11	0.11

(b) Relative Weights (Eigenvector)

The last step is to plot the weights based on the complexity (x-axis) and the severity (y-axis) values. The complexity of the pattern impacts the inspection affinity negatively while a higher severity affects the inspection affinity positively. Since we have three different categories, we divide the area into three equal divisions [55]. Figure 8 depicts the equal divisions for each category. It shows that patterns A-6 and C-6 have high affinity levels (above 60°), while pattern B-6 has a low affinity level (below 30°).

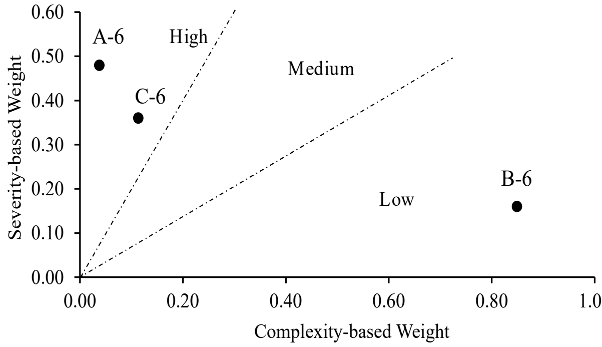


Figure 8: Communication Patterns Categorization

5. Evaluation

We tested our approach on five traces generated from three HPC systems: SMG2000 [56][57], AMG2013 [58], and the NAS BT parallel benchmark [59].

To generate traces, we instrumented the applications statically using the Score-P [60] tool, which is perhaps one of the most recommended instrumentation tools for MPI-based systems. We instrumented all the functions of a system in the same way to ensure that the added overhead, though it is known to be low with Score-P [61], is the same. This makes it possible to distinguish between normal and slow executions by factoring out the instrumentation overhead. Note that this would have been different if dynamic instrumentation is used where the trace generation is done as the system is executed, resulting in potential bias when measuring execution time. We deliberately chose static instrumentation to avoid such bias. Score-P generates traces in OTF2 [62] format. We generated the traces on a private IBM CloudBurst 2.1 cloud cluster of 14 blades.

In the following, we present an analysis for each program followed by a discussion of the approach's execution times in Section 5.4.

5.1. SMG2000

SMG2000 is a semi-coarsening multi-grid solver for linear systems [57] that performs a large number of irregular communication patterns [63]. We provide the analysis on execution traces generated using 4, 128, and 1024 processes.

5.1.1. SMG2000 with 4 Processes

We clarify the approach using a trace generated with 4 processes ($2 \times 2 \times 1$ process topology) and $10 \times 10 \times 10$ problem size. The OTF2 trace file contains a total of 6,248,096 events where 283,946 are MPI events. Figure 9 shows three different views from the trace using Vampir visualization tool [18]. Each view provides different sections of the program execution with several recurring communication patterns.

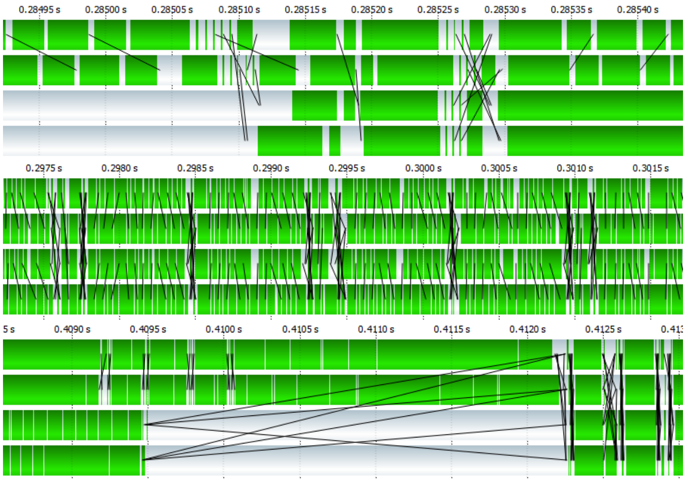


Figure 9: Zoomed-In Views using Vampir

Figure 10 shows the detected communication patterns in the trace. The trace contains 25 distinct patterns (point-to-point and collective communications) with a total of 42,222 instances. The number next to the pattern name represents its frequency. For example, CP2 is repeated 2,101 times while CP18 only appears 9 times in the trace. The name sequence is based on the order of their appearance in the trace. The trace contains three collective communication patterns (CP1, CP3, and CP25). In CP1, each process posts `ALLGATHER` and `ALLGATHERV` events during the *Initialize* phase. CP3 contains two `ALLREDUCE` events and marks the end of the main three phases (*Initialize*, *Setup* and *Solve*). CP25 is a single `ALLREDUCE` operation that occurs 7 times in the *Solve* phase.

	CP1 (1)	CP3 (2101)	CP25 (7)
P0	ALLGATHER·ALLGATHERV	ALLREDUCE·ALLREDUCE	ALLREDUCE
P1	ALLGATHER·ALLGATHERV	ALLREDUCE·ALLREDUCE	ALLREDUCE
P2	ALLGATHER·ALLGATHERV	ALLREDUCE·ALLREDUCE	ALLREDUCE
P3	ALLGATHER·ALLGATHERV	ALLREDUCE·ALLREDUCE	ALLREDUCE

	CP2 (2101)	CP4 (963)	CP5 (648)	CP6 (511)
P0	S1·S2·S3·R1·R2·R3	S1·R1		R2
P1	S0·S2·S3·R0·R2·R3	S0·R0		
P2	S0·S1·S3·R0·R1·R3		S3·R3	S0
P3	S0·S1·S2·R0·R1·R2		S2·R2	

	CP7 (511)	CP8 (9385)	CP9 (7960)	CP10 (7960)	CP11 (9385)
P0		R1			S1
P1	R3	S0			R0
P2			R3	S3	
P3	S1		S2	R2	

	CP12 (502)	CP13 (502)	CP14 (18)	CP15 (18)
P0	S2		S1·R1·R2·R3	S1·S2·S3·R1
P1		S3	S0·R0·R2·R3	S0·S2·S3·R0
P2	R0		S0·S1·S3·R3	S3·R0·R1·R3
P3		R1	S0·S1·S2·R2	S2·R0·R1·R2

	CP16 (9)	CP17 (18)	CP18 (9)	CP19 (9)	CP20 (380)
P0	S1·R1·R2·R3	S1·S2·S3·R1	R2·R3	S2·S3	S2·R2
P1	S0·R0·R2·R3	S0·S2·S3·R0	R2·R3	S2·S3	
P2	S0·S1	R0·R1	S0·S1	R0·R1	S0·R0
P3	S0·S1	R0·R1	S0·S1	R0·R1	

	CP21 (380)	CP22 (182)	CP23 (380)	CP24 (380)
P0		S1·S2·R1·R2	S1·S2·S3·R1·R2·R3	S2·S3·R2·R3
P1	S3·R3	S0·S3·R0·R3	S0·S2·S3·R0·R2·R3	S2·S3·R2·R3
P2		S0·S3·R0·R3	S0·S1·R0·R1	S0·S1·R0·R1
P3	S1·R1	S1·S2·R1·R2	S0·S1·R0·R1	S0·S1·R0·R1

Figure 10: Communication Patterns in SMG2000 ($2 \times 2 \times 1$ Topology)

Further, the trace has several point-to-point communication patterns. For example, in CP2, all the processes exchange messages with each other. On the other hand, patterns CP6–CP13 involve only one message. Patterns CP4, CP5, CP20, and CP21 involve two processes that send to and receive a message from each. Furthermore, the trace contains other patterns that involve all the processes but with different communication behaviors. For example, given the 2×2 process topology, pattern CP18 involves sending messages from the processes in the second row in the grid (P_2 and P_3) to the processes in the first row in the grid (P_0 and P_1). Similarly, in CP19, the processes in the first row in the grid send messages to the ones in the second row.

The Modified Z-Score technique detected a total of 1,995 slow pattern instances in the trace, where 1,362 were caused by late senders, and 632 from late receivers. Further, the trace contains one instance of the collective communication pattern CP3 that was slow due to late processes P_0 and P_1. Figure 11 is an excerpt of the trace which shows the first 43 patterns in the program where the slow ones are surrounded by red rectangles. The first occurrence of CP2 exchanges 41,280 bytes while the other instances in the figure exchange 1,536 and 48 bytes. We

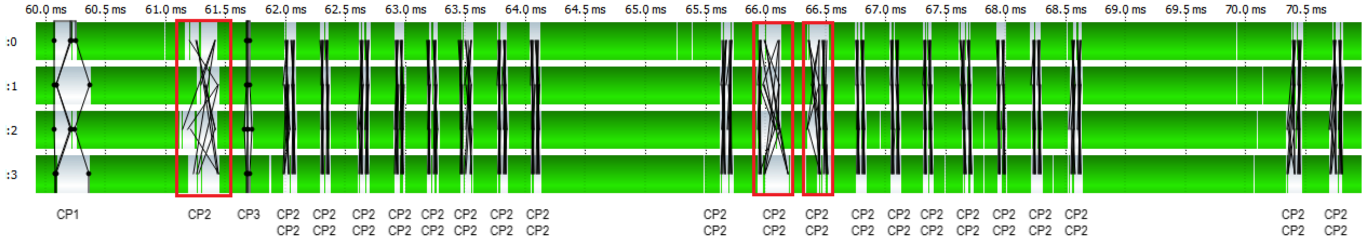


Figure 11: First 43 Patterns in the Trace (SMG2000 with 4 Processes)

only compare the instances of the same pattern that exchange the same amount of data. For example, only two instances of CP2 involve a total of 41,280 bytes in the whole trace. The first instance is at position 2 and the second is at position 1,751. The first instance was identified as slow (327,100 μs) while the second as normal (159,376 μs).

Figure 12 shows a zoomed-in view of the two slow instances of CP2 at positions 22 (48 bytes) and 23 (1,536 bytes), where the events involved in CP2 at position 22 are in red. The 48-byte instances occur 440 times in the whole trace, while the 1,536-byte instances occur 492 times. It is clear from Figure 12 that even for a small trace, without the aid of statistical techniques, visual inspection will not be practical.

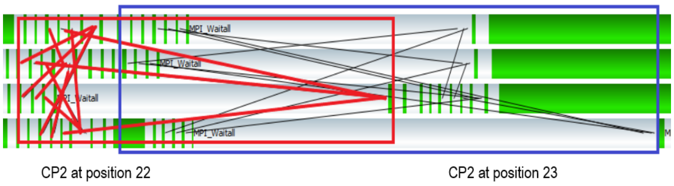


Figure 12: Slow CP2 Pattern Instances

Figure 13 shows the D_{JS} values for each pattern instance in the communication patterns sequence. Figure 13 is not concerned with time, and only depicts the D_{JS} values for each point in the sequence. The phase detection algorithm segmented the list of 42,222 communication patterns at point 7,420, which has the maximum D_{JS} value with a positive segmentation strength. This point has an instance of CP3. The first segment corresponds to the *Setup* phase while the second belongs to the *Solve* phase. The `HYPRE_StructSMGSetup` function represents the *Setup* phase and the `HYPRE_StructSMGSolve` function represents the *Solve* phase. The *Solve* phase is more homogeneous than the *Setup* phase. This is clear from the smoothness of the curve. The Setup phase has 483 slow pattern instances, while the Solve phase has 1,512 slow instances. The first slow pattern in the Solve phase is an instance of CP9 (200 bytes) at position 14 (position 7,434 in the trace). The duration of this instance is 166,000 μs while the median for CP9 (200 bytes) is 47,570 μs.

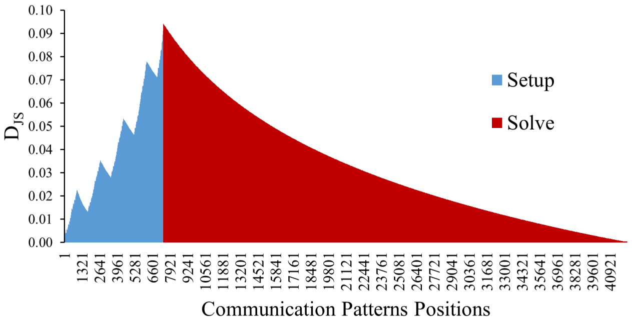


Figure 13: ND$_{JS}$ values for whole trace

Using a segmentation tree depth of 3, the number of leaf segments (phases) was 8. Table 10 shows the 8 execution phases with the corresponding number of slow communication patterns. Phase 6 is a very long phase, it stopped segmenting at depth 4 where the length of the phase was only reduced from 34,793 to 34,752. This phase is highly homogeneous, which means that it has a low level of randomness. Therefore, this phase can be further segmented into equal sized sub-phases.

Table 10: Execution Phases and Slow Communication Patterns

Phase	User Functions	From	To	Slow Patterns
1	HYPRE_StructGridAssemble HYPRE_StructMatrixAssemble hypre_SMGRelaxSetupASol	1	49	6
2	hypre_SMGRelax	50	951	99
3	hypre_SMGSetup	952	1,749	32
4	hypre_SMGSetupRAPOp hypre_SMGRelaxSetup	1,750	7,420	346
5	hypre_SMGSolve	7,421	7,423	0
6	hypre_SMGSolve	7,424	42,216	1,511
7	hypre_SMGSolve	42,217	42,219	0
8	hypre_SMGSolve	42,220	42,222	1

Table 11 shows the slow patterns in Phase 1, which are different instances of CP2. The instances at positions 22 and 24 (CP2 with 48 bytes) were identified as slow with high inspection affinity level.

Table 11: Slow CP2 instances in Phase 1 where *i*: position in phase, *l*: data length, *d*: duration, *m*: median, *t*: threshold, P_{LS}: last process to start, P_{LF}: last process to finish, IA: Inspection Affinity

i	*l* (bytes)	d (μs)	*m*	*t* (µs)	P_{LS}	P_{LF}	*IA*
2	41,280	327,100	245,438	163,776	1	3	Low
22	48	165,701	58,344	105,800	0	2	High
23	1,536	262,600	55,838	86,836	2	3	Low
24	48	135,000	58,344	105,800	3	3	High
25	1,536	100,701	55,838	86,836	3	3	Low
48	48	108,501	58,344	105,800	2	3	Medium

Further, we provide other information such as the slowest event, the shortest event, first and last processes to start, and first and last to finish in the pattern. We use the last process to start to determine whether the latency was caused by a late-sender or a late-receiver. This information when provided to the analyst will help in tracing back to the root cause. For example, CP2 at position 22 in Phase 1 was categorized with *high* inspection affinity. When investigating the trace and based on the information provided by our approach, process P_3 entered a long wait-state. The delay in this pattern was propagated to the next three successive patterns. Process P_3 waited for a long time to receive the messages from P_0, P_1 and P_2 causing a communication imbalance (see Figure 14).

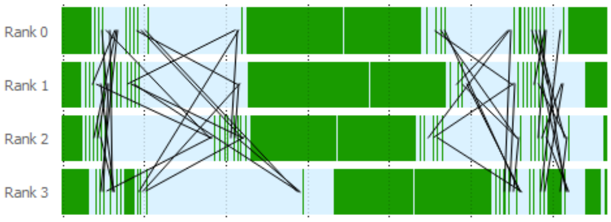


Figure 14: CP2: A wait-state causing propagation latency

Figure 15 shows a case for CP16 where the senders (P_2 and P_3) are waiting for processes P_0 and P_1 (late receivers) to complete their computation. This waiting time could be reduced with better load balancing.

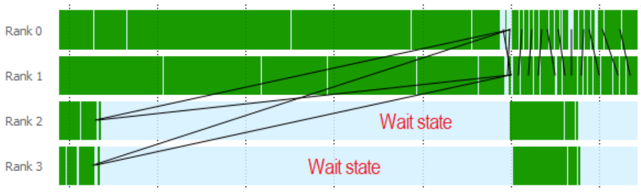


Figure 15: CP16: A wait-state from load imbalance

Figure 16 shows two instances of the collective communication pattern CP3. The first instance was identified as slow where processes P_2 and P_3 are waiting for processes P_0 and P_1 to complete their computations, which is another example of load imbalance.

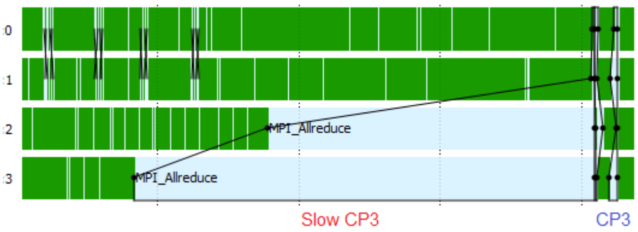


Figure 16: CP3: Slow Instance at position 7420

5.1.2. *SMG 2000 with 128 Processes*

The second scenario uses a trace generated from running SMG2000 using 128 processes. The process topology is 8×8×2 with a 1 × 1 × 1 problem size. The trace contains 14, 445, 448 events with 577, 772 MPI events.

The number of distinct communication patterns in the trace is 690 with a total of 6, 180 instances. Figure 17 presents a sample of several communication patterns in the trace, where the processes perform nearest and non-nearest neighbor communications with their partner processes in the same grid. CP166 occurred only twice, while CP188 and CP336 occurred 46 and 48 times respectively. The approach identified 609 slow pattern instances, where 514 instances were late senders, 93 were late receivers, and two instances were slow due to late processes in collective communications.

Figure 18 shows the D_{JS} values for the 6, 180 pattern list. The maximum value is at point 1, 766, which marks the end of the *Setup* phase. The communication pattern at this point is `ALLREDUCE·ALLREDUCE`, which occurs three times in the trace and marks the end of the three main phases (*Initialize*, *Setup* and *Solve*).

Table 12 shows a comparison between BIC and AIC for sequence segmentation. Using BIC for model selection, the segmentation algorithm was only able to segment the whole sequence into two phases (S_1 and S_2). However, Figure 18 shows that there is a level of randomness in the two phases which may be further segmented. AIC, on the other hand, provided deeper segmentation levels due to positive segmentation strengths. For example, the *Initialize* phase, which only includes three patterns, was distinctly identified at depth 9 in the segmentation tree.

The *Setup* phase contains 431 slow patterns while the *Solve* phase has 178 slow instances. For example, the two occurrences of CP166 (528 bytes), where the first one was marked as slow. CP188 occurred 46 times (4 in *Setup* and 42 in *Solve*). All 4 instances in the *Setup* phase were marked as slow, and only one in *Solve*. From the 48 instances of CP336 (112 bytes), only 8 occurred in the *Setup* phase (first two were marked as slow). The second instance of `ALLREDUCE·ALLREDUCE` (262, 144 bytes) was also marked as slow.

	CP166 (2)
P0	S16·S16
P1	S16·S17·S18
P2	S17·S18·S19
P3	S18·S19·S20
P4	S19·S20·S21
P5	S20·S21·S22
P6	S21·S22·S23
P7	S22·S23
P16	S32·S33·R0·R1
P17	S32·S33·S34·R0·R1·R2
P18	S33·S34·S35·R1·R2·R3
P19	S34·S35·S36·R2·R3·R4
P20	S35·S36·S37·R3·R4·R5
P21	S36·S37·S38·R4·R5·R6
P22	S37·S38·S39·R5·R6·R7
P23	S38·S39·R6·R7
P32	S48·S49·R16·R17
P33	S48·S49·S50·R16·R17·R18
P34	S49·S50·S51·R17·R18·R19
P35	S50·S51·S52·R18·R19·R20
P36	S51·S52·S53·R19·R20·R21
P37	S52·S53·S54·R20·R21·R22
P38	S53·S54·S55·R21·R22·R23
P39	S54·S55·R22·R23
P48	R32·R33
P49	R32·R33·R34
P50	R33·R34·R35
P51	R34·R35·R36
P52	R35·R36·R37
P53	R36·R37·R38
P54	R37·R38·R39
P55	R38·R39

	CP188 (46)
P32	S33
P33	R32·R34
P34	S33·S35
P35	R34·R36
P36	S35·S37
P37	R36·R38
P38	S37·S39
P39	R38

	CP336 (48)
P66	S74·R74
P74	S66·S82·R66·R82
P82	S74·S90·R74·R90
P90	S82·S98·R82·R98
P98	S90·S106·R90·R106
P106	S98·S114·R98·R114
P114	S106·S122·R106·R122
P122	S114·R114

	CP89 (18)
P41	S43
P43	R41·R45
P45	S43·S47
P47	R45

	CP401 (56)
P0	S8·R8
P8	S0·S16·R0·R16
P16	S8·S24·R8·R24
P24	S16·S32·R16·R32
P32	S24·S40·R24·R40
P40	S32·S48·R32·R48
P48	S40·S56·R40·R56
P56	S48·R48

	CP17 (2)
P71	R79
P79	S71·R87
P87	S79·R95
P95	S87·R103
P103	S95·R111
P111	S103·R119
P119	S111·R127
P127	S119

	CP364 (56)
P6	S14·R14
P14	S22·S6·R6·R22
P22	S30·S14·R14·R30
P30	S38·S22·R22·R38
P38	S46·S30·R30·R46
P46	S54·S38·R38·R54
P54	S62·S46·R46·R62
P62	S54·R54

	CP72 (4)
P25	S27·R27
P27	S25·S29·R25·R29
P29	S27·S31·R27·R31
P31	S29·R29

	CP407 (30)
P112	R113
P113	S112·S114
P114	R113·R115
P115	S114·S116
P116	R115·R117
P117	S116·S118
P118	R117·R119
P119	S118

Figure 17: Sample Patterns in SMG2000 (8 × 8 × 2 Topology)

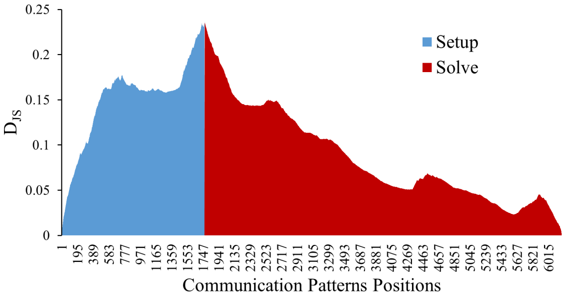


Figure 18: D_{JS} for Whole Communication Patterns List

Table 13: Execution Phases and Slow Communication Patterns

Phase	User Functions	From	To	Slow Patterns
1	HYPRE_StructGridAssemble HYPRE_StructMatrixAssemble hypre_SMGRelaxSetup	1	339	105
2	hypre_SMGRelaxSetup	340	529	45
3	hypre_SMGSetupInterpOp	530	1,453	204
4	hypre_SMGSetupRAPOp hypre_SMGRelaxSetup	1,454	1,766	77
5	hypre_SMGSolve	1,767	5,185	140
6	hypre_SMGSolve	5,186	5,908	32
7	hypre_SMGSolve	5,909	5,986	2
8	hypre_SMGSolve	5,987	6,180	4

Table 13 presents the 8 phases detected at depth 3 with the number of slow communication patterns. The analyst can further segment the long phases to reduce the number of patterns in a certain phase. Working with fine-grained views guides in the localization of the root cause.

Figure 19 shows the whole trace, using Vampir tool, anno-

Table 12: Comparison between AIC and BIC for Segmentation: starting position (p_s), ending position (p_e), segmentation point (p_{max})

S	p_s	p_e	$\hat{D}_{JS}$	p_{max}	s_{BIC}	s_{AIC}	*Parent*
S_0	1	6,180	0.24	1,766	0.13	3.19	-
S_1	1	1,766	0.30	529	-0.37	1.37	S_0
S_2	1,767	6,180	0.07	5,908	-0.42	1.45	S_0

tated with the 8 phases. Manual inspection of a trace of this size is not a trivial task. Thus, the analyst should be guided to where specific communications are happening, in which phases, and which ones are slow. Moreover, information about the severity and complexity of the slow patterns should be useful in deciding where to conduct the analysis.

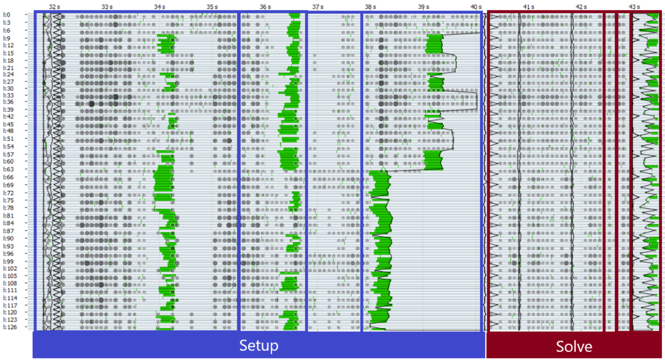


Figure 19: Execution Trace in Vampir with Phases

Figure 20 depicts the communication patterns in the execution trace where the slow ones are shown in blue. We magnified the size of slow patterns to make them clear to the reader. This applies to the other scenarios.

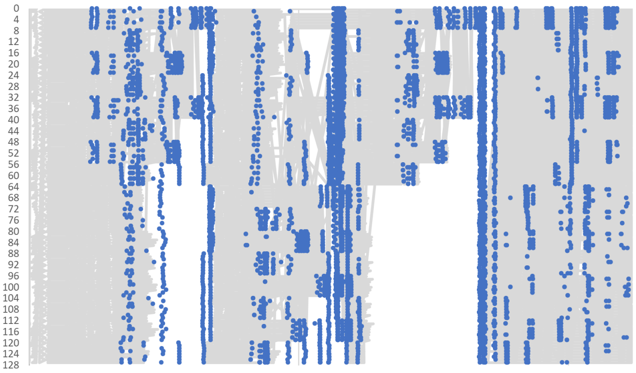


Figure 20: Communication Patterns in Execution Trace

When segmenting the trace at depth 6, the length of the first phase was reduced to 43 patterns with three slow instances. Table 14 shows that patterns CP5, CP7, and CP15 involve 32, 16, and 2 processes respectively. Our approach marked CP15 with high affinity level (due to its low complexity), CP7 with medium, and CP5 as low due to its high complexity.

Figure 21 depicts these three slow patterns. The patterns are colored based on their severity-complexity (IA) level. The light blue color depicts the normal communications in Phase 1. Communication pattern CP15 (high IA) involves processes P_{33} and P_{37}. P_{37} was the last to start in the pattern by sending its message late (i.e. late-sender). Analyzing this pattern was simple since it only involves an exchange of two messages between P_{33} and P_{37}. Further, identifying the root cause was quick, where the delay was caused by P_{37} as it was busy in computations while P_{33} was waiting for a message from P_{37}. CP7 (medium IA) involves 16 processes and 360 events, which is clearly more complex to analyze than CP15. In CP7, the last process to start was P_{37} as it was busy in computations before sending a message to P_1. Thus, P_{37} was identified as a late-sender. The slowest event was in process P_5 which was also waiting to receive from P_1. This delay caused the other processes to wait and the most impacted was P_{33} which completed its communication last. The analysis to identify the root cause in this pattern was longer than that in CP15 due to its higher complexity. The root cause was also related to computational imbalance resulting in communication imbalance due to the waiting time. Finally, CP5 which is the most complex in the phase (32 process and 888 events) was categorized as low. Understanding this communication pattern using the Vampir tool was a tedious task. The last process to start was P_{53}, where it was sending a message to P_{19}. Process P_{53} was busy in computations, which shows that the root-cause was due to computational (or load) imbalance.

Table 14: Slow Communication Patterns in Phase 1 (depth 6)
i: position in phase, *np*: number of processes, *ne*: number of events

Pattern	*i*	*np*	*ne*	*bytes*	Duration (μs)	Median	Threshold	IA
CP5	12	32	888	1,776	142,107,809	116,603,750	126,426,115	Low
CP7	22	16	360	720	120,196,558	96,675,762	73,154,966	Med.
CP15	41	2	4	32	21,438,566	6,977,436	10,924,132	High

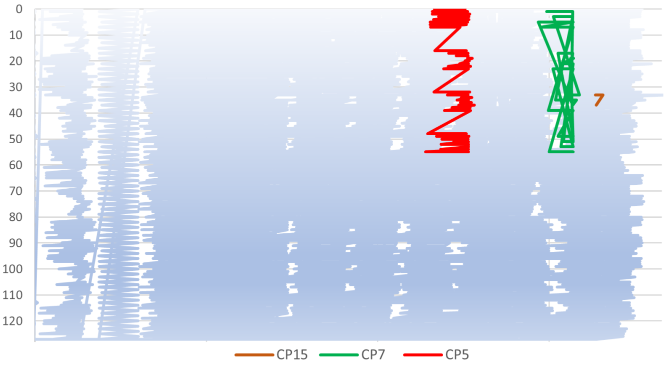


Figure 21: Communication Patterns in Execution Trace

The trace also contains 96 patterns that occurred only once (where 80 patterns consist of a single message communication). The remaining 16 patterns involve different number of processes (8, 16, 32, 64, and 128) that communicate with their nearest and non-nearest neighboring processes. For example, pattern CP139 involves all the processes in the program, where each process in the first 2D grid (processes 0 to 63) sends to its neighbors in the same grid and to its neighbors in the second grid (processes 64 to 127). However, the processes in the second grid only receive data from their adjacent neighbors in the first grid. In this paper, we only identify slow patterns that repeat in the trace. However, it is important to identify if non-repeating communication patterns are causing latency. These patterns can be compared to similar repeating and non-repeating patterns in terms of number of processes, events, and size of exchanged data.

5.1.3. SMG2000 with 1024 Processes

We also tested our approach on a trace of 1,024 processes using a $16 \times 16 \times 4$ process topology with a $1 \times 1 \times 1$ problem size. The Total number of OTF2 events is 598,699,396 with 11,866,079 MPI events. The number of detected distinct communication patterns is $5,777$ with a total of $64,052$ instances. Our approach identified $3,512$ slow pattern instances, where 2,967 instances were due to late senders, 544 by late receivers, and two were from late processes in collective communications. There are also $2,122$ non-repeating patterns in the trace where $1,568$ are composed of a single message and 512 are composed of 3 messages. The remaining 42 communications involve different number of processes ranging from 16 to $1,024$.

Figure 22 shows sample nearest and non-nearest neighbor communication patterns that are repeating in the trace. The trace contains many point-to-point patterns that involve various number of processes (ranging from 2 to 1,024 processes).

	CP974 (90)
P32	R33
P33	S32·S34
P34	R33·R35
P35	S34·S36
P36	R35·R37
P37	S36·S38
P38	R37·R39
P39	S38·S40
P40	R39·R41
P41	S40·S42
P42	R41·R43
P43	S42·S44
P44	R43·R45
P45	S44·S46
P46	R45·R47
P47	S46

	CP2210 (80)
P257	S273·R273
P273	S257·S289·R257·R289
P289	S273·S305·R273·R305
P305	S289·S321·R289·R321
P321	S305·S337·R305·R337
P337	S321·S353·R321·R353
P353	S337·S369·R337·R369
P369	S353·S385·R353·R385
P385	S369·S401·R369·R401
P401	S385·S417·R385·R417
P417	S401·S433·R401·R433
P433	S417·S449·R417·R449
P449	S433·S465·R433·R465
P465	S449·S481·R449·R481
P481	S465·S497·R465·R497
P497	S481

	CP3008 (90)
P897	S901
P901	R897·R905
P905	S901·S909
P909	R905

	CP38 (12)
P225	S229.R229
P229	S225.S233.R225.R233
P233	S229.S237.R229.R237
P237	S233.R233

Figure 22: Sample Patterns in SMG2000 ($16 \times 16 \times 4$ Topology)

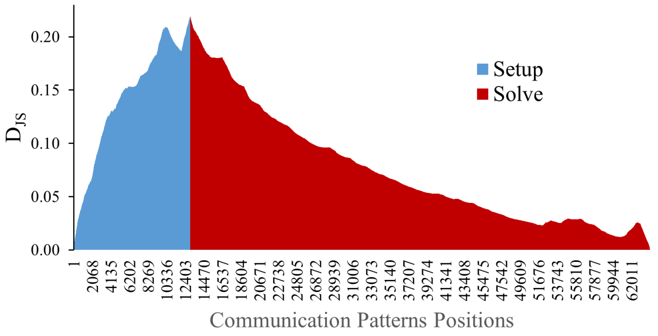


Figure 23: D_{JS} for Whole Communication Patterns List

Figure 23 depicts the D_{JS} values for the whole sequence of communication patterns in the trace. The behavior is similar to the previous two scenarios. The segmentation is performed at point $12,961$ which marks the end of the *Setup* phase.

Table 15 shows the 8 phases at depth 3 in the segmentation tree with the user functions that they occur in and the number of slow patterns in each phase.

Table 15: Execution Phases and Slow Communication Patterns

Phase	User Functions	From	To	Slow Patterns
1	HYPRE_StructGridAssemble HYPRE_StructMatrixAssemble hypre_SMGRelaxSetup	1	3,086	610
2	hypre_SMGRelaxSetup	3,087	8,183	515
3	hypre_SMGSetupInterpOp	8,184	9,818	146
4	hypre_SMGSetupRAPOp hypre_SMGRelaxSetup	9,819	12,960	760
5	hypre_SMGSolve	12,961	59,438	1,463
6	hypre_SMGSolve	59,439	62,707	4
7	hypre_SMGSolve	62,708	62,976	7
8	hypre_SMGSolve HYPRE_StructSMGDestroy	62,977	64,052	7

Figure 24 depicts the communication patterns and the main two phases in the trace, where slow patterns are in darker blue.

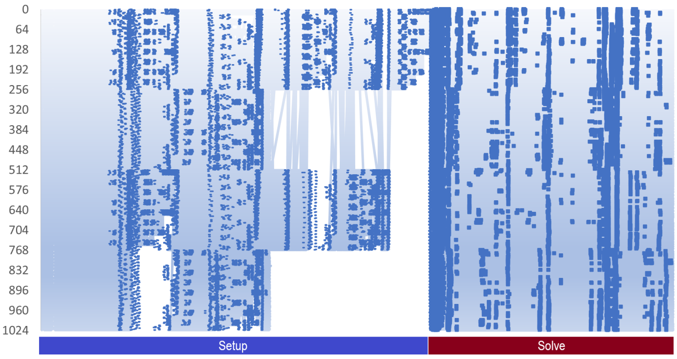


Figure 24: Communication Events in Execution Trace

Increasing the segmentation tree depth increases the number of phases and results in shorter ones. Table 16 shows the slow patterns in Phase 8 and Phase 24 when segmenting the trace at depth 6. In Phase 8, three slow patterns were categorized with low, medium and high IA levels. Slow instance of pattern CP1751 at position 106 was categorized as *high* since it only consists of two events (low complexity). This pattern repeats 24 times in the trace, where process P_{337} sends a message to P_{345}. For this instance, P_{345} was identified as a late-receiver. When investigating the trace, we found that the delay propagated from the preceding pattern (another instance of CP1751), causing process P_{345} to enter a wait state. Analyzing the previous instance of CP1751 showed that P_{345} was busy in computations while process P_{337} was idle (computation imbalance). Further, the analysis of pattern CP1420 (low IA) in this phase showed that process P_{375} was late in sending a message to P_{374} due to computation imbalance.

In CP2122, process P_9 was a late-sender to P_{25}. Investigating this pattern was more complex than CP1420 since it involved more events. The latency in P_9 propagated to the other processes in the pattern which caused P_{153} to finish sending its message late to process P_{169}. In Phase 24, all the patterns have equal number of processes and events. Pattern CP677 was categorized as *high* due to its high severity, where process P_{600} was identified as a late-sender when sending a message to P_{584}. Further, the analysis of slow communication patterns in this scenario ($16 \times 16 \times 4$ process topology) showed that the latency was also related to a communication issue, which could be a result of resource limitation due to the large number of processes and excessive communications.

Table 16: Slow Patterns in Phases 8 and 24 (depth 6)
i: position in phase, *np*: number of processes, *ne*: number of events

	Pattern	*i*	*np*	*ne*	*bytes*	Dur. (μs)	Median	Threshold	IA
Phase 8	CP1420	50	16	30	120	24,569,225	8,371,079	11,323,924	Low
	CP1751	106	2	2	8	8,711,713	3,793,115	5,895,232	High
	CP2122	258	16	60	240	342,098,199	16,613,767	31,253,557	Med.
Phase 24	CP640	29	2	2	8	1,466,800	1,341,800	1,216,800	Low
	CP652	44	2	2	8	545,726	437,495	329,264	Low
	CP677	62	2	2	8	6,528,768	5,690,704	4,852,640	High
	CP691	70	2	2	8	3,949,956	3,773,906	3,597,856	Med.
	CP702	74	2	2	8	3,502,378	3,260,228	3,018,078	Med.
	CP813	102	2	2	8	3,024,405	3,018,423	3,012,441	Med.
	CP833	110	2	2	8	2,557,835	2,508,315	2,458,795	Med.

When using a modified Z-Score of $M_i > 5$, the number of slow patterns was reduced to $2,502$. This enables the analyst to focus on the most severe patterns first. The analyst can adjust this number based on his findings and can select which category of slow patterns to show. The analyst should navigate through the patterns based on the happened-before relationship and trace back to the root cause of the problem.

5.2. AMG2013

We applied our approach on the AMG2013 parallel benchmark [58]. AMG2013 examines parallel weak scaling efficiency. It is an algebraic multigrid solver for linear systems occurring in problems on unstructured grids. AMG2013 uses both MPI and OpenMP to achieve parallelism (SPMD). In our experiments, we generated the traces using 64 processes ($4 \times 4 \times 4$ topology), $10 \times 10 \times 10$ problem size and `OMP_NUM_THREADS=1`.

The trace contains a total of $19,452,036$ events ($1,879,455$ MPI events). It contains 56 distinct communication patterns with a total of 728 instances (where 30 have single occurrences and involve all the processes in the trace with various number of events). The communication patterns are mostly global (involve all the processes in the program). For example, CP4 repeats 111 times in the trace where the processes perform nearest neighbor communications in the same grid and the adjacent grids. AMG2013 has the *Setup* (`HYPRE_PCGSetup`) and *Solve* (`HYPRE_PCGSolve`) as its main phases.

Figure 25 presents the D_{JS} values for segments S_0, S_2, S_6, and S_8 in the tree. It shows that S_2 was divided into two segments (S_5 and S_6), and S_6 was segmented into two phases where S_8 corresponds to the *Solve* phase. The latter was distinctly identified at depth 3 in the segmentation tree (S_0 is at depth 0). Figure 25 shows that the *Solve* phase is highly homogeneous which is depicted by the low level of divergence.

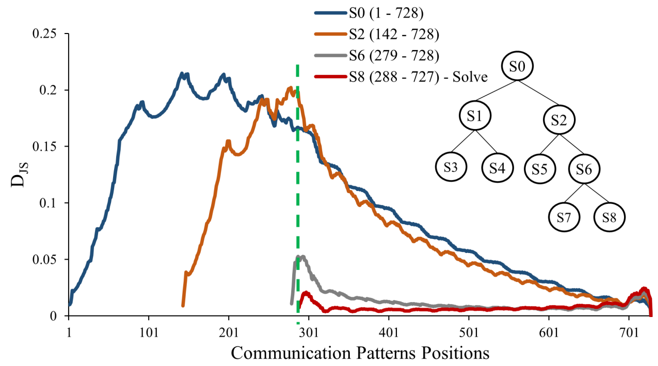


Figure 25: ND$_{JS}$ values for whole trace

Figure 26 shows the whole execution trace captured using the Vampir tool with annotations of the 8 identified phases. The *Setup* phase contains 7 sub-phases where each phase corresponds to specific user functions in the trace.

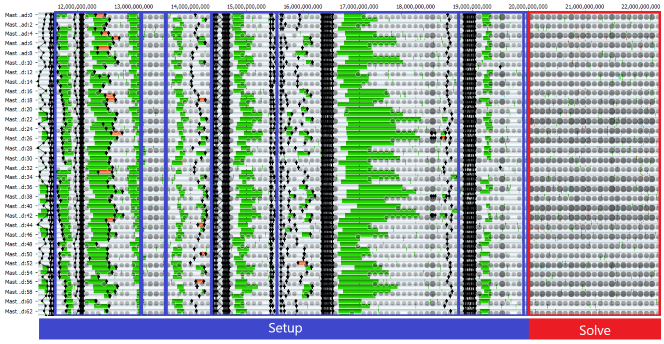


Figure 26: Execution Trace in Vampir with Phases

The Modified Z-Score method identified 65 instances as slow, where 48 were caused by late senders, one instance by late receivers, and 16 from slow processes in collective communications. The Setup phase, which is 287 patterns long, has 46 slow instances, while the *Solve* phase (441 patterns long) has 19 slow instances. Table 17 shows the 8 phases at depth 3, their user functions, and the corresponding slow patterns. The first phase contains 36 communication patterns where 12 of them were identified as slow.

Figure 27 depicts the communication patterns in the trace, where the slow communications are in blue and the normal communication events are in light gray.

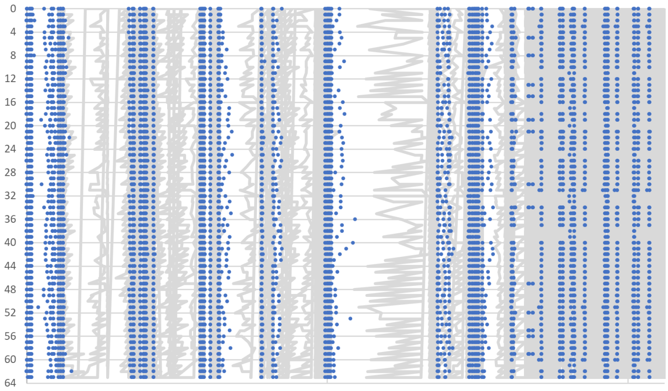


Figure 27: Communication Events in Execution Trace

Table 17: Execution Phases and Slow Communication Patterns

Phase	User Functions	From	To	Slow Patterns
1	hypre_BoomerAMGCoarsenHMIS hypre_BoomerAMGCreate2ndS hypre_BoomerAMGCoarsenRuge	1	36	12
2	hypre_BoomerAMGCoarsenPMIS hypre_BoomerAMGBuildMultipass	37	86	7
3	hypre_BoomerAMGBuildCoarseOp	87	91	0
4	hypre_BoomerAMGCoarsenHMIS hypre_BoomerAMGBuildExtPIInterp	92	142	9
5	hypre_BoomerAMGBuildCoarseOp hypre_BoomerAMGCoarsenHMIS hypre_BoomerAMGBuildExtPIInterp	143	201	7
6	hypre_BoomerAMGBuildCoarseOp hypre_BoomerAMGCoarsenHMIS hypre_BoomerAMGBuildExtPIInterp	202	278	11
7	hypre_BoomerAMGBuildCoarseOp hypre_BoomerAMGCoarsenHMIS hypre_BoomerAMGBuildExtPIInterp	279	287	0
8	HYPRE_PCGSolve	288	728	19

Table 18 shows the 12 slow communication patterns in Phase 1. They are point-to-point and collective communications that involve 64 processes (i.e. global). The patterns that involve higher number of events have higher complexity. CP3 and CP4 are point-to-point communication patterns. CP3 is a large pattern that repeats 4 times in the trace with 2 instances marked as slow. CP4 repeats 111 times with 9 slow instances in the trace. The whole trace contains 4 instances of CP1, a collective communication pattern (ALLREDUCE·ALLREDUCE), where only its first instance (which is also the first communication in the trace) was identified as slow. The program also contains 128 instances of ALLREDUCE (CP2), where 11 of them are slow (3 slow in Phase 1). CP4 at positions 6 and 7 both have process P_{61} as a late-sender when it was sending a message to P_{45}. CP4 at position 6 was categorized as medium while the instance at position 7 was categorized as low due to its lower severity. When analyzing the trace, the cause of delay was related to computational imbalance as the late processes were busy in computations while the others were set idle.

Table 18: Slow Communication Patterns in Phase 1

Ptrn	*i*	*np*	*ne*	*bytes*	Duration (μs)	Median	Threshold	IA
CP1	1	64	128	65,536	239,226,449	8,420,955	9,569,933	High
CP2	2	64	64	32,768	49,099,716	8,517,267	24,026,711	High
CP2	3	64	64	32,768	31,699,789	8,517,267	24,026,711	High
CP3	4	64	1,404	20,736	26,272,627	19,157,037	12,041,447	Low
CP3	5	64	1,404	232,704	218,539,989	115,149,039	11,758,089	Low
CP4	6	64	576	230,400	388,590,357	11,078,898	21,154,570	Med.
CP4	7	64	576	230,400	155,497,529	11,078,898	21,154,570	Low
CP2	8	64	64	32,768	106,645,449	8,517,267	24,026,711	High
CP4	24	64	576	115,200	13,115,945	5,910,651	6,759,955	Low
CP4	25	64	576	115,200	13,311,987	5,910,651	6,759,955	Low
CP4	32	64	576	115,200	16,648,752	5,910,651	6,759,955	Low
CP4	33	64	576	115,200	140,555,667	5,910,651	6,759,955	Med.

5.3. NAS BT Parallel Benchmark

We tested our approach on a trace generated from the Block Tri-diagonal solver (BT) pseudo application from the NAS parallel benchmarks suite [59]. This suite contains a small set of applications for performance evaluation of parallel supercomputers. We generated a trace from running BT on 100 processes (10 × 10 process topology) using the A class configuration. We applied a Score-p filter to exclude some utility functions to reduce the trace file size (`binvcrhs*, matmul_sub*, matvec_sub*, exact_solution*, binvrhs*, lhs*init*, timer_*`). The trace contains 24,454,798 events with 4,605,900 as MPI events.

Our approach detected 64 distinct communication patterns with a total of 108,745 instances. The trace contains $3,283$ slow patterns, where only one was in a collective communication and the rest $3,282$ were caused by late senders. The program performs one BCAST, two ALLREDUCE, and one REDUCE collective operations. The first communication in the trace is the BCAST while the ALLREDUCE and REDUCE are the last communications respectively. The trace contains a global point-to-point communication pattern (CP2), that repeats 202 times, where all the processes send messages to and receive from their left, right, upper, lower and anti-diagonal neighbors. There are also 60 short point-to-point communication patterns, where each pattern repeats 1,809 times in the trace. They belong to six different groups that occur in a specific solve function. Figure 28 shows one sample pattern from each group.

	CP7		**CP15**		**CP25**
P0	S1·R9	P0	S9·R1	P0	S10·R90
P1	S2·R0	P1	S0·R2	P10	S20·R0
P2	S3·R1	P2	S1·R3	P20	S30·R10
P3	S4·R2	P3	S2·R4	P30	S40·R20
P4	S5·R3	P4	S3·R5	P40	S50·R30
P5	S6·R4	P5	S4·R6	P50	S60·R40
P6	S7·R5	P6	S5·R7	P60	S70·R50
P7	S8·R6	P7	S6·R8	P70	S80·R60
P8	S9·R7	P8	S7·R9	P80	S90·R70
P9	S0·R8	P9	S8·R0	P90	S0·R80

	CP33		**CP47**		**CP58**
P0	S90·R10	P0	S19·R91	P0	S91·R19
P10	S0·R20	P19	S28·R0	P19	S0·R28
P20	S10·R30	P28	S37·R19	P28	S19·R37
P30	S20·R40	P37	S46·R28	P37	S28·R46
P40	S30·R50	P46	S55·R37	P46	S37·R55
P50	S40·R60	P55	S64·R46	P55	S46·R64
P60	S50·R70	P64	S73·R55	P64	S55·R73
P70	S60·R80	P73	S82·R64	P73	S64·R82
P80	S70·R90	P82	S91·R73	P82	S73·R91
P90	S80·R0	P91	S0·R82	P91	S82·R0

Figure 28: Local Communication Patterns

In the following, we describe each group and show the user function it occurs in.

- G1 (x_solve_): The processes in each row (x-axis) send to the right and receive from the left neighbor (e.g. CP7).
- G2 (x_solve_): The processes in each row send to the left and receive from the right neighbor (e.g. CP15).
- G3 (y_solve_): The processes in each column (y-axis) send to the lower and receive from the upper neighbor (e.g. CP25).
- G4 (y_solve_): The processes in each column send to the lower and receive from the upper neighbor (e.g. CP33).
- G5 (z_solve_): The processes in each anti-diagonal (z-axis) send to the lower-left neighbor and receive from the upper-right neighbor (e.g. CP47).

- G6 (z_solve_): The processes in each anti-diagonal send to the upper-right neighbor and receive from the lower-left neighbor (e.g. CP58).

Figure 29 shows a screenshot from the Vampir tool for a repeating communication pattern (same as CP15) in the trace.

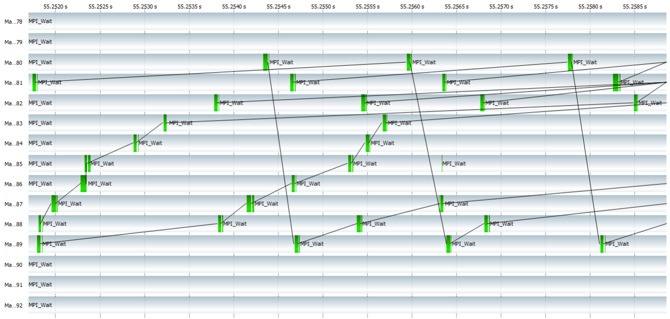


Figure 29: Zoomed-in view using Vampir

Figure 30 shows the D_{JS} values for the whole trace. The trace contains a long homogeneous *Solve* phase which is demonstrated by the smoothness of the curve.

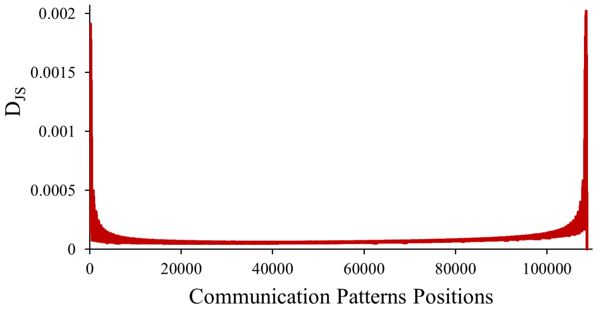


Figure 30: Jensen-Shannon Divergence at S_0

Table 19 shows four phases detected at depth 2 in the segmentation tree. Phase 1 involves one instance of the `copy_faces_` and `x_solve_` functions. The second phase is a long phase that should be further segmented. The third and fourth phases involve the `z_solve_` and `copy_faces_` functions. The pattern at point 1 is a `BCAST` operation, followed by CP2 (occurs in copy_faces_), and 180 short point-to-point patterns. Similarly, phases 3 and 4, together contain 180 short patterns followed by CP2, two `ALLREDUCE` operations, and one `REDUCE` operation.

Table 19: Phases at segmentation depth 2

Phase	User Functions	From	To	Slow Patterns
1	copy_faces_ x_solve_	1	182	49
2	x_solve_ y_solve_ z_solve_ copy_faces_	183	108,562	3,232
3	z_solve_ copy_faces_	108,563	108,652	1
4		108,653	108,746	1

To further segment Phase 2, we set the *length* parameter to 200 to prevent segmenting any phase that is less than 200 patterns. We selected this value based on the length of Phase 1 which belongs mainly to the x_solve_ function. Table 20 presents the sub-phases of Phase 2. It shows how the segmentation approach is capable of identifying distinct short phases in the trace. This long phase will be subsequently segmented to detect finer sub-phases, which is different than the *Solve* phases in SMG2000 and AMG2013 where they stopped segmenting due to negative segmentation strength.

Table 20: Sub-phases in Phase 2

Sub-Phase	User Functions	From	To	Slow Patterns
2.1	y_solve_	183	362	51
2.2	z_solve_ copy_faces_	363	543	1
2.3	x_solve_	544	723	0
2.4	y_solve_	724	903	
2.5	z_solve_ copy_faces_	904	1,084	13
2.6	x_solve_ y_solve_ z_solve_ copy_faces_	1,085	108,562	3,165

Figure 31 depicts the communication events in the whole execution trace. The normally behaving patterns are in light blue, while the slow events are in darker blue. It shows that there are repeating waves of slow events in the trace.

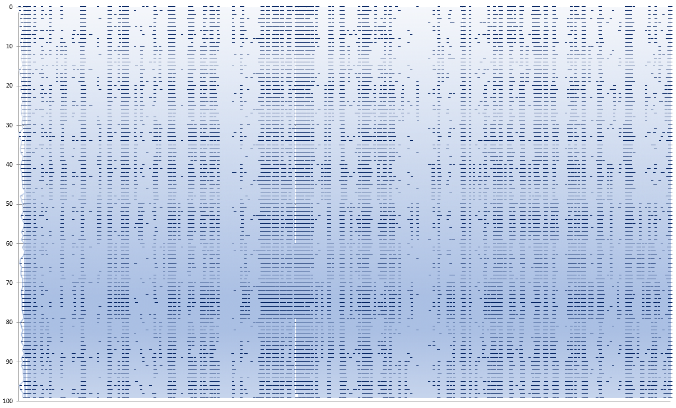


Figure 31: Communication Events in Execution Trace

Our approach is capable of providing finer views of the trace at deeper segmentation levels. The recognition of phases at this level of granularity makes it easier to analyze the trace and compare the patterns in the phase. For example, Table 21 shows the slow communication patterns in Phase 6 when segmenting the trace at depth 8. Even though the patterns have the same number of processes, events, and data length, they have been categorized differently based on their severity level (duration). The severity in CP9 was clear when analyzing the trace, where process P_{47} was the last to start when sending a message to P_{48}. Process P_{47} was busy in computations which caused the pattern to be slower than its peers in the trace. For CP4, there was lack of synchronization between the processes due to load imbalance as some processes were busy in computations while the others were set idle.

Table 21: Slow Communication Patterns in Phase 6 at depth = 8
i: position in phase, processes = 10, events = 20, data length = 117600 bytes, P_{LS} last process to start, and P_{LS} last to finish

Pattern	i	Duration (μs)	Median	Threshold	P_{LS}	P_{LF}	IA
CP4	1	75,980,041	32,944,462	68,698,667	27	27	Low
CP7	2	90,115,100	34,156,897	72,057,660	3	3	Low
CP12	3	165,094,010	31,797,205	64,371,365	97	97	Medium
CP9	4	517,507,257	33,601,163	72,984,113	47	47	High
CP4	5	72,882,120	32,944,462	68,698,667	27	27	Low
CP8	6	281,978,202	33,067,682	73,696,175	88	88	Medium
CP10	7	329,193,924	34,084,422	72,353,890	61	61	Medium
CP4	8	74,083,586	32,944,462	68,698,667	27	27	Low

5.4. Execution Time Analysis of the Approach

We tested our approach on a Windows 10 running on a 2.4 GHz Intel Core i7-3517U with 8GB RAM. We measured the execution time for each of the following steps in the approach using the five scenarios.

- *Step 1:* Building the maximal repeats table per process
- *Step 2:* Communication patterns construction
- *Step 3:* Localization of patterns (phase detection)
- *Step 4:* Inefficient patterns identification
- *Step 5:* Inefficient patterns categorization

Table 22 shows each scenario, its pattern length, the number of distinct patterns, and the execution time for each step. The time for phase detection depends on the length of the sequence of patterns and the number of distinct patterns. For example, SMG2000 with 1,024 processes contains 5,777 distinct patterns (parameters) with a total length of 64,052. It spent much more time than the SMG2000 with 4 processes (25 patterns and 42,222 sequence length). Moreover, it took much longer than the NAS BT case which has a length of 108,745 patterns with 64 distinct ones.

Table 22: Execution Times (ms) for each step at depth 3
Number of phases = 8

	SMG2000 (4)	*SMG2000* (128)	*SMG2000* (1024)	*AMG2013* (64)	*NAS BT* (100)
length	42,222	6,180	64,052	728	108,745
patterns	25	690	5,777	56	64
Step 1 (ms)	221	325	4,507	437	934
Step 2 (ms)	466	991	25,990	2,037	10,854
Step 3 (ms)	670	2,450	219,506	55	5,291
Step 4 (ms)	106	46	219	35	284
Step 5 (ms)	548	78	928	9	3,803

Table 23 shows the execution times for steps 3 and 5 at various depths. The phase detection times slightly changed due to the shorter phases at deeper depths in the trees, and that some phases stopped segmenting due to their high homogeneity. The pattern categorization times showed an increase for the first four traces, and were almost unchanged in case of NAS BT.

5.5. Threats to Validity

Internal validity threats concern the factors that might influence our results. The selection of the systems used in the evaluation section is one possible threat. These systems are used by similar studies and are representative of typical HPC systems. Another threat to internal validity is the way we evaluated the patterns that we detected as inefficient. Although every effort was made to provide a thorough evaluation, errors may have occurred. Finally, errors in the implementation of our tool may be a threat to internal validity. To mitigate this threat, we tested the tool. We also make the tool available for other researchers to review.

Table 23: Execution times and number of phases
for variable depths

	Depth	*SMG2000* (4)	*SMG2000* (128)	*SMG2000* (1024)	*AMG2013* (64)	*NAS BT* (100)
No. of Phases	6	38	51	61	49	50
	9	96	374	478	104	132
	12	176	1,349	3,535	129	212
Step 3 (ms)	6	822	2,845	270,032	64	7,489
	9	850	2,852	271,577	70	11,054
	12	888	2,940	293,161	74	11,225
Step 5 (ms)	6	565	68	704	22	4,007
	9	595	166	1,047	45	3,970
	12	622	417	2,186	59	4,038

Conclusion Validity: Conclusion validity threats depend on the correctness of the results. We made every effort to review the results obtained from our experiments to ensure that we properly evaluate the effectiveness of our approach. We strive to provide as many details as possible to allow the evaluation and reproducibility of our results.

Reliability Validity: Reliability validity corresponds to the possibility of replicating this study. We examined five traces from three different HPC systems. These systems were also used by other studies. We acknowledge that we need more traces to strongly support our findings. This said, we put online material to allow other researchers to reproduce the study.

External Validity: External validity is related to the generalizability of the results. We experimented with three HPC systems that use MPI for inter-process communication. Our approach relies solely on MPI calls and no other mechanism. As such, we believe that it should work on any system that uses MPI for inter-process communication. This said, we agree that generalization remains an external threat to validity of our study. We can see a situation where an HPC system uses other communication mechanisms. For that, we will need to adapt our approach to traces generated from these systems.

6. Conclusion

We presented a novel approach for the discovery of slow communication patterns in execution traces using statistical analysis techniques. Our approach can also be used for program comprehension to help analysts understand the behavior of the inter-process communication in MPI programs. Our approach is built on an improved version of our previous trace segmentation approach, which uses information theory principles to split a trace on execution phases. The approach relies on the detection of communication patterns in a trace. For this, we used routine-call trees to identify the maximal repeats in each process. Each phase contains a list of slow communication patterns categorized based on their severity and complexity. We also used a median-based statistical approach to identify the outlier patterns in the trace. The Modified Z-score is a robust statistical method that is not greatly affected by outliers.

We tested our approach on five traces from three open HPC programs that use MPI for inter-process communication. The results show that our approach can accurately identify execution phases based on the detected communication patterns, is capable of identifying slow patterns within each phase, and determine whether the latency is caused by late senders or receivers. These inefficient patterns are categorized based on their severity and complexity levels to help an analyst make decisions on where to start the inspection.

Our approach suffers from the existence of non-repeating communications in the program execution in some scenarios. These communications may hold important information regarding the performance of the program. In this paper, we did not target the identification of latency in non-repeating communications and their significance on the performance of HPC systems.

In the future, we intend to apply various similarity techniques to compare single occurring communications to other patterns in the trace in order to identify whether they are slow or not. We will also extend the trace segmentation approach to avoid the problem of long homogeneous segments caused by the context sensitivity problem [64] instead of segmenting them into sub-phases of equal length.

Further, We want to apply our approach to HPC systems that use other communication mechanisms to generalize our approach to work on traces generated from these systems.

Finally, we intend to conduct a user study to provide a qualitative evaluation of the effectiveness of our approach. We are also investigating the application of our approach to industrial systems.

7. Reproduction Package

The implementation of our approach as well as the data used in the evaluation section are made available in the following repository: https://github.com/lalawneh/HPC-MPI-Traces.

Acknowledgment

The authors would like to thank the Deanship of Research at Jordan University of Science & Technology for funding this research (ID. 20200285). Also, the first author, Dr. Luay Alawneh, would like to thank Concordia University, Canada, where this research was conducted during his sabbatical leave.

References

[1] S. Heldens, P. Hijma, B. V. Werkhoven, J. Maassen, A. S. Belloum, R. V. Van Nieuwpoort, The landscape of exascale research: A data-driven literature analysis, ACM Computing Surveys (CSUR) 53 (2) (2020) 1–43.

[2] Mpi 3.0: Message passing interface version 3.0.
URL `https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`

[3] J. Navaridas, J. Miguel-Alonso, F. J. Ridruejo, On synthesizing workloads emulating mpi applications, in: 2008 IEEE International Symposium on Parallel and Distributed Processing, IEEE, 2008, pp. 1–8.

[4] R. Preissl, T. Köckerbauer, M. Schulz, D. Kranzlmüller, B. R. de Supinski, D. J. Quinlan, Detecting patterns in mpi communication traces, in: 2008 37th International Conference on Parallel Processing, IEEE, 2008, pp. 230–237.

[5] M. Casas, R. M. Badia, J. Labarta, Automatic phase detection and structure extraction of mpi applications, The International Journal of High Performance Computing Applications 24 (3) (2010) 335–360.

[6] K. E. Isaacs, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, P.-T. Bremer, Ordering traces logically to identify lateness in message passing programs, IEEE Transactions on Parallel and Distributed Systems 27 (3) (2015) 829–840.

[7] L. Alawneh, A. Hamou-Lhadj, J. Hassine, Segmenting large traces of inter-process communication with a focus on high performance computing systems, Journal of Systems and Software 120 (2016) 1–16.

[8] H. Pirzadeh, S. Shanian, A. Hamou-Lhadj, L. Alawneh, A. Shafiee, Stratified sampling of execution traces: Execution phases serving as strata, Science of Computer Programming 78 (8) (2013) 1099–1118.

[9] G. L. T. Chetsa, L. Lefevre, J.-M. Pierson, P. Stolf, G. Da Costa, A user friendly phase detection methodology for hpc systems' analysis, in: 2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing, IEEE, 2013, pp. 118–125.

[10] C. E. Shannon, A mathematical theory of communication, The Bell system technical journal 27 (3) (1948) 379–423.

[11] I. Grosse, P. Bernaola-Galván, P. Carpena, R. Román-Roldán, J. Oliver, H. E. Stanley, Analysis of symbolic sequences using the jensen-shannon divergence, Physical Review E 65 (4) (2002) 041905.

[12] H. Akaike, Likelihood of a model and information criteria, Journal of econometrics 16 (1) (1981) 3–14.

[13] B. Iglewicz, D. C. Hoaglin, How to detect and handle outliers, Vol. 16, Asq Press, 1993.

[14] T. L. Saaty, How to make a decision: the analytic hierarchy process, European journal of operational research 48 (1) (1990) 9–26.

[15] F. Darema, The spmd model: Past, present and future, in: European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting, Springer, 2001, pp. 1–1.

[16] G. Schwarz, Estimating the dimension of a model, The annals of statistics (1978) 461–464.

[17] W. Li, P. Bernaola-Galván, F. Haghighi, I. Grosse, Applications of recursive segmentation to the analysis of dna sequences, Computers & chemistry 26 (5) (2002) 491–510.

[18] Vampir: Visualization and analysis of mpi resources.
URL `https://vampir.eu/`

[19] S. S. Shende, A. D. Malony, The tau parallel performance system, The International Journal of High Performance Computing Applications 20 (2) (2006) 287–311.

[20] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, W. E. Nagel, The vampir performance analysis tool-set, in: Tools for high performance computing, Springer, 2008, pp. 139–155.

[21] D. Böhme, F. Wolf, B. R. de Supinski, M. Schulz, M. Geimer, Scalable critical-path based performance analysis, in: 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IEEE, 2012, pp. 1330–1340.

[22] M. Schulz, Extracting critical path graphs from mpi applications, in: 2005 IEEE International Conference on Cluster Computing, IEEE, 2005, pp. 1–10.

[23] H. Wei, J. Gao, P. Qing, K. Yu, Y.-F. Fang, M.-L. Li, Mpi-rcdd: A framework for mpi runtime communication deadlock detection., J. Comput. Sci. Technol. 35 (2) (2020) 395–411.

[24] G. Mao, D. Böhme, M.-A. Hermanns, M. Geimer, D. Lorenz, F. Wolf, Catching idlers with ease: A lightweight wait-state profiler for mpi programs, in: Proceedings of the 21st European MPI Users' Group Meeting, 2014, pp. 103–108.

[25] A. Sikora, T. Margalef, J. Jorba, Automated and dynamic abstraction of mpi application performance, Cluster Computing 19 (3) (2016) 1105–1137.

[26] X. Aguilar, K. Fürlinger, E. Laure, Automatic on-line detection of mpi application structure with event flow graphs, in: European Conference on

Parallel Processing, Springer, 2015, pp. 70–81.

[27] E. Jeannot, R. Sartori, Improving mpi application communication time with an introspection monitoring library, in: 21st IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2020), 2020, p. 10.

[28] S. Ramesh, A. Mahéo, S. Shende, A. D. Malony, H. Subramoni, A. Ruhela, D. K. D. Panda, Mpi performance engineering with the mpi tool interface: the integration of mvapich and tau, Parallel Computing 77 (2018) 19–37.

[29] S. Taheri, I. Briggs, M. Burtscher, G. Gopalakrishnan, Difftrace: Efficient whole-program trace analysis and diffing for debugging, in: 2019 IEEE International Conference on Cluster Computing (CLUSTER), IEEE, 2019, pp. 1–12.

[30] E. Gallardo, J. Vienne, L. Fialho, P. Teller, J. Browne, Employing mpi_t in mpi advisor to optimize application performance, The International Journal of High Performance Computing Applications 32 (6) (2018) 882–896.

[31] J. P. Kenny, K. Sargsyan, S. Knight, G. Michelogiannakis, J. J. Wilke, The pitfalls of provisioning exascale networks: A trace replay analysis for understanding communication performance, in: International Conference on High Performance Computing, Springer, 2018, pp. 269–288.

[32] B. Mohr, F. Wolf, Kojak–a tool set for automatic performance analysis of parallel programs, in: European Conference on Parallel Processing, Springer, 2003, pp. 1301–1304.

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

[34] D. Böhme, M. Geimer, L. Arnold, F. Voigtlaender, F. Wolf, Identifying the root causes of wait states in large-scale parallel applications, ACM Transactions on Parallel Computing (TOPC) 3 (2) (2016) 1–24.

[35] X. Wu, F. Mueller, Scalaextrap: Trace-based communication extrapolation for spmd programs, ACM SIGPLAN Notices 46 (8) (2011) 113–122.

[36] M. Tsuji, T. Boku, M. Sato, Scalable communication performance prediction using auto-generated pseudo mpi event trace, in: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, 2019, pp. 53–62.

[37] S. Miwa, I. Laguna, M. Schulz, Predcom: A predictive approach to collecting approximated communication traces, IEEE Transactions on Parallel and Distributed Systems 32 (1) (2020) 45–58.

[38] A. Knüpfer, B. Voigt, W. E. Nagel, H. Mix, Visualization of repetitive patterns in event traces, in: International Workshop on Applied Parallel Computing, Springer, 2006, pp. 430–439.

[39] F. Trahay, E. Brunet, M. M. Bouksiaa, J. Liao, Selecting points of interest in traces using patterns of events, in: 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, IEEE, 2015, pp. 70–77.

[40] T. Kunz, M. F. Seuren, Fast detection of communication patterns in distributed executions, in: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research, IBM Press, 1997, p. 12.

[41] T. Köckerbauer, C. Klausecker, D. Kranzlmüller, Scalable parallel debugging with g-eclipse, in: Tools for High Performance Computing 2009, Springer, 2010, pp. 115–123.

[42] J. Gonzalez, J. Gimenez, J. Labarta, Automatic detection of parallel applications computation phases, in: 2009 IEEE International Symposium on Parallel & Distributed Processing, IEEE, 2009, pp. 1–11.

[43] J. Gonzalez, K. Huck, J. Gimenez, J. Labarta, Automatic refinement of parallel applications structure detection, in: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IEEE, 2012, pp. 1680–1687.

[44] M. Casas, R. M. Badia, J. Labarta, Automatic phase detection of mpi applications, Parallel Computing: Architectures, Algorithms, and Applications 38 (2007) 129–136.

[45] D. Gusfield, Algorithms on stings, trees, and sequences: Computer science and computational biology, Acm Sigact News 28 (4) (1997) 41–60.

[46] G. Schwarz, et al., Estimating the dimension of a model, The annals of statistics 6 (2) (1978) 461–464.

[47] G. d. N. P. Leite, A. M. Araújo, P. A. C. Rosas, T. Stosic, B. Stosic, Entropy measures for early detection of bearing faults, Physica A: Statistical Mechanics and its Applications 514 (2019) 458–472.

[48] M. Dabbagh, S. P. Lee, R. M. Parizi, Functional and non-functional requirements prioritization: empirical evaluation of ipa, ahp-based, and ham-based approaches, Soft computing 20 (11) (2016) 4497–4520.

[49] C. A. Tu, E. Rasoulinezhad, T. Sarker, Investigating solutions for the development of a green bond market: Evidence from analytic hierarchy process, Finance Research Letters 34 (2020) 101457.

[50] P. Aragonés-Beltrán, F. Chaparro-González, J.-P. Pastor-Ferrando, A. Pla-Rubio, An ahp (analytic hierarchy process)/anp (analytic network process)-based multi-criteria decision approach for the selection of solar-thermal power plant investment projects, Energy 66 (2014) 222–238.

[51] P. H. Dos Santos, S. M. Neves, D. O. Sant'Anna, C. H. de Oliveira, H. D. Carvalho, The analytic hierarchy process supporting decision making for sustainable development: An overview of applications, Journal of cleaner production 212 (2019) 119–138.

[52] A. Ruhela, H. Subramoni, S. Chakraborty, M. Bayatpour, P. Kousha, D. K. Panda, Efficient asynchronous communication progress for mpi without dedicated resources, in: Proceedings of the 25th European MPI Users' Group Meeting, 2018, pp. 1–11.

[53] I. Abraham, T. H. Chan, D. Dolev, K. Nayak, R. Pass, L. Ren, E. Shi, Communication complexity of byzantine agreement, revisited, in: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, 2019, pp. 317–326.

[54] F. Cappello, O. Richard, D. Etiemble, Understanding performance of smp clusters running mpi programs, Future Generation Computer Systems 17 (6) (2001) 711–720.

[55] A. van Lamsweerde, Requirements engineering: From system goals to uml models to software specifications (2009).

[56] P. N. Brown, R. D. Falgout, J. E. Jones, Semicoarsening multigrid on distributed memory machines, SIAM Journal on Scientific Computing 21 (5) (2000) 1823–1834.

[57] Advanced simulation and computing program: The asc smg 2000 benchmark code.
URL `http://tau.uoregon.edu/tau-wiki/SMG2000`

[58] Parallel algebraic multigrid solver for linear systems: The asc amg 2013 benchmark code.
URL `https://computing.llnl.gov/projects/co-design/amg2013`

[59] Nas parallel benchmarks.
URL `https://www.nas.nasa.gov/publications/npb.html`

[60] Scalable performance measurement infrastructure for parallel codes.
URL `https://www.vi-hps.org/projects/score-p/`

[61] J. Eriksson, P. Ojeda-May, T. Ponweiser, T. Steinreiter, Profiling and tracing tools for performance analysis of large scale applications, PRACE: Partnership for Advanced Computing in Europe (2016).

[62] D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, F. Wolf, Open trace format 2: The next generation of scalable trace formats and support libraries., in: PARCO, Vol. 22, 2011, pp. 481–490.

[63] J. Vetter, Dynamic statistical profiling of communication activity in distributed applications, ACM SIGMETRICS Performance Evaluation Review 30 (1) (2002) 240–250.

[64] S.-A. Cheong, P. Stodghill, D. J. Schneider, S. W. Cartinhour, C. R. Myers, The context sensitivity problem in biological sequence segmentation, arXiv preprint arXiv:0904.2668 (2009).