A Contextual Approach for Effective Recovery of Inter-process Communication Patterns from HPC Traces

[1,2]Luay Alawneh, [1]Abdelwahab Hamou-Lhadj, [1]Syed Shariyar Murtaza, and [1]Yan Liu
[1]Software Behaviour Analysis (SBA) Research Lab
Department of Electrical and Computer Engineering
Concordia University, Montreal, QC, Canada
[2]Jordan University of Science & Technology,
Faculty of Computer & IT, Software Engineering Department, Jordan
{l_alawne, abdelw, smurtaza}@ece.concordia.ca and {yan.liu@concordia.ca}

***Abstract*—Studies have shown that understanding of inter-process communication patterns is an enabler to effective analysis of high performance computing (HPC) applications. In previous work, we presented an algorithm for recovering communication patterns from traces of HPC systems. The algorithm worked well on small cases but it suffered from low accuracy when applied to large (and most interesting) traces. We believe that this was due to the fact that we viewed the trace as a mere string of operations of inter-process communication. That is, we did not take into account program control flow information. In this paper, we improve the detection accuracy by using function calls to serve as a context to guide the pattern extraction process. When applied to traces generated from two HPC benchmark applications, we demonstrate that this contextual approach improves precision and recall by an average of 56% and 66% respectively over the non-contextual method.**

***Index Terms*—Dynamic Analysis, High Performance Computing, Inter-Process Communication Patterns, Reverse Engineering.**

I. INTRODUCTION

A confluence of trends has led to an increasing demand for High Performance Computing (HPC) systems. The need to process complex scientific problems in record time plays an important role. The emergence of new and powerful computing platforms is also a significant factor.

While HPC systems offer tremendous benefits to users, they pose real challenges to developers. They tend to be difficult to debug and analyze. HPC systems differ from traditional software in terms of their form, structure, and behaviour [7]. Typical systems involve a large number of processes interacting with each other following various communication patterns (see next section for examples).

As an HPC application undergoes several ad-hoc maintenance activities, it becomes challenging to know which communication patterns the application supports. Without this knowledge, it is difficult for software engineers to debug and analyze the application. Communication patterns are useful in revealing the program's parallel structure and communication topology [2], which can in turn help in many software maintenance tasks including debugging [22], refactoring and optimization [18]. Communication patterns can also be used to validate whether the actual program behaves according to the intended design or not. Rendered as event graphs (see Figures 1 and 2 for examples), they are used as documentation that guides further changes made to the system.

It is therefore essential to investigate techniques and tools that can automatically identify communication patterns from HPC systems. In previous work, we presented an approach that aims to do just that by mining execution traces [1]. The approach, however, suffers from low accuracy, especially when applied to large systems. After a careful analysis of several HPC traces using trace analyzers such as the Vampir trace analysis tool [28], we found that this limitation was mainly due to the fact that we viewed a trace as a mere stream of MPI[1] events independently from where these events appear in the program. We observed that patterns rarely appear outside the boundaries of user-defined program functions. We therefore decided to investigate if and how user-defined functions can be used as a context that guides the detection of communication patterns. We also propose an improved pattern detection algorithm that scales up to so large traces. This is important because of the size overhead added when tracing user-defined functions along with MPI calls.

When applied to traces generated from two HPC systems, we found that the contextual approach (presented in this paper) performs in average 56% to 66% better than the previous approach without penalizing performance.

The remaining parts of the paper are organized as follows: In Section II, we provide background information on HPC systems. In Section III, we present our approach and discuss its components. In Section IV, we show the effectiveness of our approach by applying it to traces generated from two HPC systems. We continue the paper by presenting the threats to validity in Section V. Related work is described in Section VI. We conclude the paper in Section VII.

[1] MPI stands for Message Passing Interface, a standard for inter-process communication [8].

II. Background

HPC benefits from parallel computing in order to solve computation-intensive scientific problems. As opposed to sequential computing, parallel computing decomposes the problem into sub-problems that run on different computational units in order to solve the problem in a reasonable amount of time. In most cases, the computational units need to collaborate in order to complete a specific task. This collaboration is achieved either through shared memory or inter-process communication. In the latter case, which is the focus of this paper, typical HPC programs may involve a large number of processes that coordinate their efforts to solve a specific large scale problem. This is performed usually through well-known communication patterns.

An example of a communication pattern is shown in Figure 1. The figure depicts a sample trace generated from running four processes in parallel. A horizontal line represents the events from a particular process. When matching the MPI events on the partner processes, a communication pattern will be generated. The figure shows a 2D-nearest-neighbour communication pattern (with a 4 x 1 process topology) that is repeated three times at different locations in the graph. Non-MPI events are represented using dark bars. The graph that we used to depict the communication events is called the event graph [16] where time is on the x-axis and the events flow from left to right.

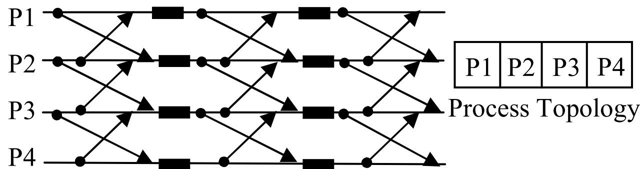

Fig. 1. Repeating Communication Pattern

A process topology is the way the processes are represented on a grid (Cartesian) or a graph structure. For example, in Figure 1 (right), the process topology is a 4x1 grid. The same system can have different process topologies. For example, a 4x4 grid topology will have 16 processes that are arranged on a square grid. Similarly a 4x4x2 grid topology would involve processes that are arranged on two superposed 2D, forming a 3D grid.

When detecting communication patterns, we are interested in the way the program processes are communicating and not what data they are exchanging. For example, each pattern in Figure 1 may involve different data (messages exchanged among processes) but the processes are still communicating based on the same pattern.

Several communication patterns for distributed systems are reported in the literature [21]. They are used as guidelines for the proper way to implement an inter-process communication mechanism (for more details about the list of documented communication patterns, please refer to [21]). Figure 2 depicts an example of two communication patterns used in implementing collective communications. The Binary Tree pattern (Figure 2a) is used to implement All-to-One MPI collective operations. For example, the MPI_Reduce operation is implemented using this pattern. The Butterfly pattern, shown in Figure 2b, is a communication pattern that is used to implement All-to-All MPI collective operations.

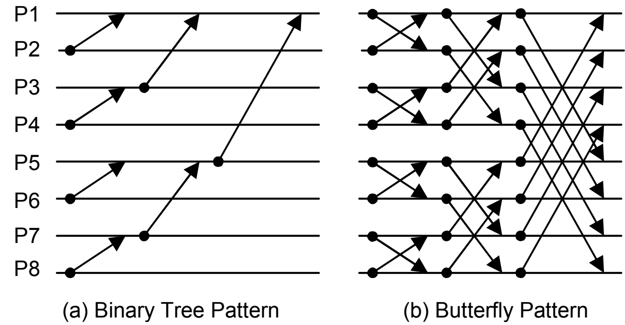

Fig. 2. Examples of communication patterns

A simple way to recover communication patterns is to consider each process trace as a string of MPI operations (e.g., send, receive, etc.) and apply a pattern recognition algorithm (see [1] for an example). The main advantage of this approach is that it only needs to trace the MPI operations. This results in a smaller trace than a trace containing other types of information such as routine calls. Also, many HPC trace analyzers offer views that only handle inter-process communication operations, making them a good candidate for supporting this method.

However, this approach (as we will show in the case studies) results in a large number of patterns among which many of them are false positives. They do not represent the communication pattern intended during the design. This approach also results in overlapping patterns, making it difficult to know the beginning and end of patterns.

Sequence: (mirrors a process trace generated from Sweep3D)
abababacacacbdbdbdadadadabababacacacbdbdbdadadadabababacac acbdbdbdadadad

Detected Patterns: Number in brackets shows how many times the pattern occurs in the sequence above.

a (27), aba (9), ababa (6), abababacacacbdbdbdadadad (3), ac (9), acac (6), b (18), bd (9), bdbd (6), da (11), dad (9), dada (8), dadad (6), dadada (5), d (18), abababacacacbdbdbdadadadabababacacacbdbdbdadadad (2)

Valid Patterns: ab (9), ac (9), bd (9), ad (9), abababacacacbdbdbdadadad (3)

Fig. 3. Pattern Detection Based on Syntactic Methods

Some of these limitations can be illustrated in the example of Figure 3, which is taken from a real system execution. The presented sequence simulates a large trace that is generated from running the Sweep3D [26] HPC program. We denote the MPI events using alphabet characters to avoid clutter. The communication patterns intended for this application are documented in [26]. Sweep3D implements a wavefront pattern with a sweep from each corner in the process topology to its opposite corner. The example shows that 16 patterns were detected when using a technique that only uses MPI operations

despite the fact that only five patterns are valid patterns according to the system's documentation. In addition, the approach missed two valid patterns 'ab' and 'ad'. It is clear that a better and more accurate approach is needed.

III. Approach

Our approach for detecting communication patterns in HPC traces is shown in Figure 4. For simplicity reasons, we refer to a trace of one process as a process trace and we use the term 'trace' alone when referring to the whole program trace (multiple processes). We refer to patterns that are repeated in one process trace as process patterns. A communication pattern is then a collection of process patterns.

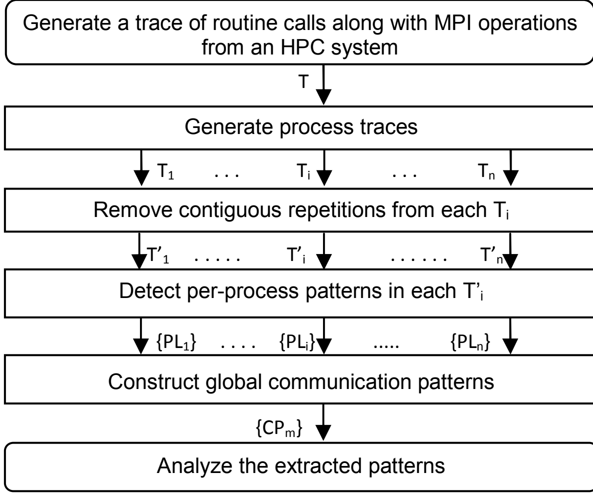

Fig. 4. Pattern Detection Approach

The first step of our approach is to collect traces of routine calls along with MPI operations and divide them into process traces. Then, we remove contiguous repetitions to reduce the trace size. The next two steps, which represent the core of the approach, focus on the detection of communication patterns. The final step is a validation step in which we semi-automatically analyze the quality of the extracted patterns. We discuss each step in more details in the subsequent sections.

A. Trace Generation

We generate traces using source code instrumentation by automatically inserting probes in places of interest. There exist several tools that automatically instrument MPI applications such as VampirTrace [28]. An alternative approach, in the absence of the source code, would be to instrument the operating environment such as the operating system or the virtual machine. This technique usually has a lower overhead on the system than a pure application instrumentation technique. An example of an efficient tool that could be used for Linux-like systems is LTTng (Linux Tracing Toolkit) [5].

We generate a trace by exercising a specific scenario and providing the input parameters. HPC applications also necessitate that we specify the number of processes and process topology. We can vary the process counts to increase or decrease the processing power. This is constrained by the capabilities of the host node. An example of a process topology is a 2x3 topology, which means that there are 6 processes mapped to a 2x3 matrix. A process topology can also be three dimensional.

Once the trace is generated, we create a process trace for each process by simply parsing the trace and saving process information in different files.

B. Removal of Contiguous Repetitions

Contiguous repetitions of events (also called tandem repeats) exist in a trace mainly due to the presence of loops, recursion, or the way the system is exercised. The size of the trace can be significantly reduced by removing the contiguous repeats before the pattern detection process takes place as noted by Hamou-Lhadj in [10]. Also, we found that another advantage of removing contiguous repeats is that it enables the detection of patterns in their compact form. For example, 'ababcdcdefef' can be represented as 'abcdef', which is more reflective in the cases we saw than having repetitions within a pattern. We do not keep track of the number of tandem repeats.

C. Detection of Communication Patterns

This phase is composed of two steps. The first one is to detect patterns that appear within each process. The second step is to assemble these patterns to form the final communication patterns using a pattern construction algorithm. This two-step process was first introduced by Preissl et al. [22]. It is intended to simplify the detection mechanism; it would be unnecessarily complex to proceed in one phase.

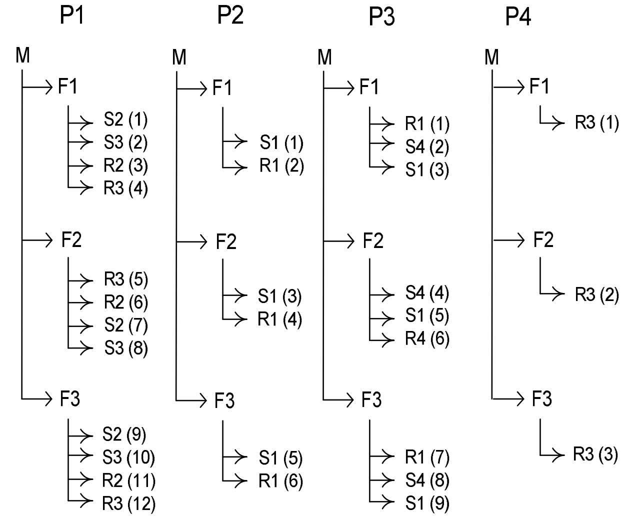

Fig. 5. Sample Traces from four Parallel Processes

We use the fictive trace illustrated in Figure 5 as a running example to explain the pattern detection algorithm. In this example, we have four processes that communicate with each other to execute functions F1 to F3. The MPI operations used in this example are simple Send (represented by S) and Receive (represented by R). The numbers between brackets represent the position of an MPI operation in a process trace. They are added here for clarity. In this example, Process P1

executes function M which calls F1, F2, and F3. In F1, P1 sends a message to Process P2, then sends a message to P3, etc. We only consider Send and Receive operations for simplicity reasons but in fact any MPI operation can be used. The messages and the timestamps are omitted from this figure.

Step 1: Detection of per-process patterns

In this step, we detect patterns of MPI operations in each process separately. A pattern is a maximal repeat that can be extended neither to the left nor to the right. More formally, given a string *S* of length *n*, a maximal repeat in S is a tuple *(p1, p2, l)* such that:

$$\exists S[p_1 .. p_1 + l - 1] = S[p_2 .. p_2 + l - 1] \text{ and } p_2 > p1 \text{ and } S[p_1 + l] \neq S[p_2 + l] \text{ and } S[p_1 - 1] \neq S[p_2 - 1]$$

The pattern detection algorithm we propose in this paper is an improvement of an earlier algorithm that we presented in [1]. The algorithm is based on the concept of n-gram extraction. In this new version, we use a bi-grams table of all bi-gram occurrences in the trace. This helps in eliminating the positions in the trace that do not correspond to repeating patterns. In addition, we perform forward and reverse pattern lookup as opposed to the previous method where we only used reverse pattern lookup. Also, we modified the previous algorithm to take into account the function calls, which are now used as a context that delimits the scope of a pattern. That is, we do not consider patterns that are formed across functions. Note that the algorithm is performed recursively over the call tree all sub-trees.

We recognize that determining a pattern's context is not that straightforward. In theory, there might be situations where the context spreads over multiple cohesive functions. However, in most benchmark systems that we have analyzed, such as the ones presented in the case study, we found that by simply considering each function as a unique context, we could significantly improve the accuracy of the pattern detection process. Perhaps, this is due to the fact that, unlike traditional systems, HPC systems (more particularly the ones used for scientific computations) rely on a set of key functions that handle most of the computations; each function is dedicated to one important cohesive computation [7].

However, given the noticeable increase in using the HPC paradigm in other application domains, triggered mainly by the emergence of multi-core and cloud computing platforms, we anticipate that HPC systems will increase in complexity. It is therefore recommended that future work should be directed towards investigating ways to determine the context (e.g., using clustering or other similar techniques). We also suggest that tools that support the analysis of HPC traces provide the users with enough flexibility to define the context, for example, by grouping functions using domain knowledge. We believe that this domain knowledge (if available) can further enhance the pattern detection process.

Another change we made to the previous algorithm is to improve its performance by adopting an n-gram construction process borrowed from the Lempel-Ziv-Welch data compression algorithm [29]. The new algorithm performs reverse and forward lookup of the pattern table, which reduces the number of lookups as opposed to the old way. In the new algorithm, as we move forward in the trace we already have the patterns detected as a result of performing forward lookup. The details of the algorithms are presented in the next paragraphs. A performance gain is particularly important since the approach presented in this paper relies on additional information (functions calls), which result in larger traces than when simply processing MPI events. Other pattern detection algorithms (e.g., [1, 14, 22]) can also be used as long as they can scale up to large traces.

The algorithm presented in this paper starts by identifying the bi-grams that are repeated more than once in a process trace and saves them in a hash table along with their positions. Again, the events of an n-gram *must* appear within the same function call since the function calls are used as a context.

The position of an n-gram is the position of its first MPI event. We also keep the frequency (note that the frequency can be inferred from the positions -we used it here for illustration purposes). We only save the bi-grams that appear more than once since these are the ones with the potential of forming a pattern. For example, the bi-gram table of process P1 is:

2-gram	**Position**	**Frequency**
S2S3	1, 7, 9	3
S3R2	2, 10	2
R2R3	3, 11	2

The next step is to take each bi-gram, appends to it the next event appearing in the trace, and check if the resulting n-gram also appears in the other positions. For example, the resulting 3-grams that is formed by appending the next event that appears in the trace to bi-grams S2S3 appearing at positions 1, 7, and 9 are S2S3R2, S2S3 (no more MPI events are found in the function), and S2S3R2 respectively. We create a 3-gram table based on that by again keeping the position and the frequency of the new formed 3-grams. We also update the bi-gram table by modifying the frequency of the sequences that contributed to the formation of a larger pattern (in this case S2S3R2). The frequency of S2S3 in the bi-gram table will be changed to 1 since two of its occurrences led to the formation of a larger pattern. The 3-gram table is shown below. It should be noted that, in practice, we do not need to keep multiple tables; it is sufficient to use one table and augment the algorithm with simple checks.

3-gram	**Position**	**Frequency**
S2S3R2	1, 9	2
S3R2R3	2, 10	2

By continuing this process, another repetitive 4-gram is formed, which is S2S3R2R4. No more n-grams can be constructed from this since we depleted all the events in the functions. The final step is to revisit the tables, take the n-grams that have a frequency greater than 1, and prune the ones that have fully served in the formation of larger n-grams. For example, both occurrences of R2R3 were used to form S2R2R3. We do not need to keep R2R3.

The final patterns that are extracted from processes P1, P2, P3, and P4 are as follows:

P1: PT1 = S2S3R2R3
P2: PT2 = S1R1
P3: PT3 = R1S4S1
P4: No patterns

It should be noted that we do not need to keep track of patterns of length 1 (such as the case of R3 in P4) to save processing time. These patterns will be integrated during the construction of global communication patterns anyway as we will show in Step 2.

Step 2: Formation of final communication patterns

Once all per-process patterns are detected, we assemble them to form the final communication patterns. The construction process iterates through the per-process patterns and maps the MPI operations with the ones of the partner processes. For a pattern p_1, its corresponding partners are those that have matching events with p_1. For example, the matching events of PT1: S2S3R2R3 are R1 in Process P2 (to match S2), R1 in Process P3 to match S3, S1 in P2 to match R2, and finally S1 in P3 to match R3. The construction process continues until all MPI operations are matched. Note that, sometimes, we might not find the matching event since we do not keep track of single events as part of the repeats (e.g., R3 in P4). In this case, we simply add it to the construction process. The final global communication pattern of the trace in Figure 5 is shown in Figure 6.

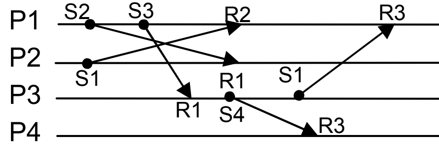

Fig. 5. Detected communication pattern

D. Validating the Patterns

In this step, we verify the accuracy of the detected patterns. This step is done semi-automatically using a trace analysis tool. We map the patterns to the original trace to make sure that all contributing processes are covered. We check that the routines are indeed responsible for the patterns. We do this by referring to the source code or any available documentation. We use documented systems in our case studies to validate the results in the absence of software engineers. If the patterns are not correct then we examine the causes by further exploring the trace. Sometimes, the cause might be due to some noise in the data such as the presence of utility routines. The objective of this step is also to learn about ways to improve the approach for future studies.

IV. Case Study

We show the effectiveness of our approach by applying it to two HPC benchmark systems, Sweep3D [26] and SMG2000 [25]. We compare the approach presented in this paper (that we refer to as ***Contextual Technique***) with the one presented in our previous work which does not use any contextual information [1] (we refer to it as ***Non-Contextual Technique***). Because the contextual approach uses additional information (i.e., function calls), we also examine the time efficiency of both approaches.

It is worth noting that we also attempted to compare our method with other researchers' approaches. The closest study is the one by Preissl et al. [22, 23]. The authors proposed an approach for detecting communication patterns using the concept of seed functions augmented with static analysis (see Related Work). The authors' work was described in a less granular manner, limiting our ability to make direct comparison between the two approaches.

A. Description of the Target Systems

Sweep3D [26] is part of the ASCI Blue Benchmark Suite. It models a 3D discrete ordinates neutron transport and represents the heart of a real ASCI application. Sweep3D includes the streaming and the scattering operators. The streaming operator is solved by sweeps from each angle to the opposite angle in the grid. The scattering operator is solved iteratively.

SMG2000 [25] is a parallel semi-coarsening multi-grid solver for the linear systems arising from finite difference, finite volume, or finite element discretization of the diffusion equation on logically rectangular grids. It uses a complex communication pattern [9]. The parallelism in SMG2000 is achieved by data decomposition. SMG2000 performs a large number of non-nearest-neighbour point-to-point communication operations and can be considered a stress test for the network subsystems of a machine as shown in [30].

B. Quality of Pattern Detection

We use precision and recall to measure the effectiveness of both approaches, which we define as follows. Both metrics vary from 0 (worst results) to 100% (best results).

$$Precision = \frac{Number\ of\ valid\ patterns\ that\ are\ detected}{Total\ number\ of\ all\ detected\ patterns}$$

$$Recall = \frac{Number\ of\ valid\ patterns\ that\ are\ detected}{Total\ number\ of\ valid\ patterns\ of\ the\ system}$$

Sweep3D: by referring to Sweep3D documentation, we found that the system uses a series of wavefront communication patterns as the main inter-process communication mechanism. We used this information to validate our results.

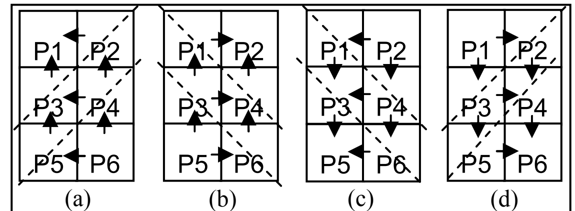

Fig. 7. Wavefront Pattern (2x3 Process Topology)

Figure 7 shows the four sweeps in a 2x3 process topology. Each sweep sends data from a corner to its opposite corner in the grid. In case of a 3D grid, Sweep3D employs eight sweeps (originating from each corner) per iteration.

We tested our approach on six traces generated from running the program using different process topologies and a number of iterations (iteration is just an input parameter for the program, needed to solve the problem implemented in Sweep3D). In all cases, Sweep3D had the same communication behaviour, i.e., wavefront pattern. The global communication pattern (composition of all wavefront patterns) was repeated the same number of times as the number of iterations. Table 1 shows the result of comparing the two approaches on a trace from a 2x3 process topology.

TABLE 1. NUMBER OF DETECTED COMMUNICATION PATTERNS IN A SWEEP3D TRACE (2X3 TOPOLOGY AND 12 ITERATIONS)

Pattern Detection Technique	Patterns	Precision	Recall
Non-Contextual	2	100%	40%
Contextual	4	100%	80%

As shown in Figure 8, the total number of communication patterns in Sweep3D is four wavefront patterns and a global communication pattern which can be seen as the composition of these four patterns. In total, there are five valid patterns.

Table 1 shows that when using the non-contextual approach only two valid communication patterns were detected. The first pattern is the one shown in Figure 8a and the second one is the global communication pattern (the composition of the four wavefronts). The precision of the non-contextual approach in this scenario was 100% however the recall was only 40%.

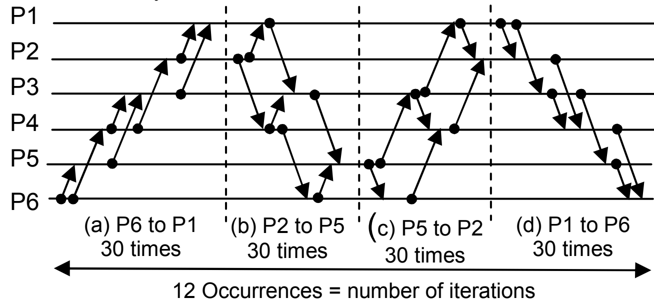

Fig. 8. Detected Communication Patterns

When applying the contextual approach, we were able to detect all the wavefront patterns. However, the global communication pattern was not detected. This is expected because we only detected patterns within specific functions. To detect the global communication pattern (i.e., the repetition of the four patterns), we would need to change the context. This is like considering the function M (Figure 5 Process P1) as a context instead of F1, F2, and F3. The precision for the contextual approach is 100% and recall of 80%.

SMG2000: We tested our approach on a trace generated from exercising a scenario with an 8x1x1 process topology and a 2x2x2 problem size. Based on the system documentation, SMG200 supports eight communication patterns (see Figure 9). We used this knowledge to validate our approach.

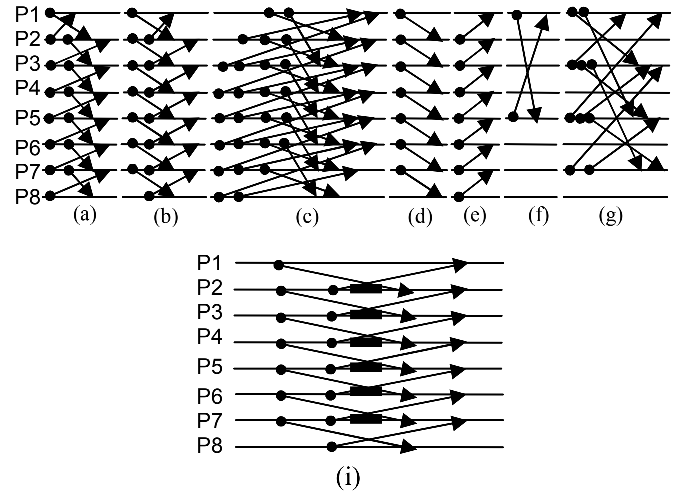

Fig. 9. SMG2000 Communication Patterns (Topology: 8x1x1 - Problem Size: 2x2x2)

Table 2 shows the results of applying the contextual and non-contextual approaches. When applying the non-contextual approach, the total number of detected patterns was 42 with only four valid communication patterns (9a, 9b, 9d and 9e) - a precision of around 10% (4/42) and a recall of 50% (4/8).

Using the contextual approach, the number of detected patterns was reduced to nine, among which we detected seven valid patterns - precision of 78% (7/9) and recall of 87% (7/8). It should be noted that the difference in results obtained between analyzing Sweep3D and SMG2000 can be explained by the fact that these systems vary in complexity. Sweep3D uses only one communication patterns whereas SMG2000 uses eight.

TABLE 2. NUMBER OF DETECTED COMMUNICATION PATTERNS FOR SMG2000 TRACE (8X1X1 PROCESS TOPOLOGY; 2X2X2 INPUT)

Pattern Detection Technique	Patterns	Precision	Recall
Non-Contextual	42	10%	50%
Contextual	9	78%	87%

The pattern that was not detected is Pattern 9(i) (Figure 9). We analyzed one of the process traces to see where the algorithm failed. We picked Process P2. We found that the function hypre_InitializeCommunication was called twice by hypre_SMGResidual and later by the function hypre_BoxGetStrideSize. In both cases, the first occurrence of hypre_InitializeCommunication called S3S1 whereas the second occurrence called R1R3 MPI operations. We treated these two calls as independent calls, which led to two different contexts. Had we consider the same function as one context (independently from the MPI calls it makes) we would have detected the pattern. This is an interesting case which clearly stresses the need for further studies to investigate a better way of defining a context than considering each function as a unique context.

In another scenario where we used a 2x2x2 process topology (3D mesh) and a 2x2x2 problem size, we were able to detect another set of patterns that the non-contextual approach was not able to detect. Table 3 shows the precision and recall values for both approaches. In this scenario, there are 13 valid communication patterns. In the non-contextual approach, we were able to detect seven valid patterns out of the 50 detected patterns, whereas using the contextual approach we were able to detect 12 valid patterns out of 17. Again, the pattern that was not detected was due to the hypre_InitializeCommunication function.

TABLE 3. NUMBER OF DETECTED COMMUNICATION PATTERNS FOR SMG2000 TRACE (2X2X2 PROCESS TOPOLOGY; 2X2X2 INPUT)

Pattern Detection Technique	Patterns	Precision	Recall
Non-Contextual	50	14%	55%
Contextual	17	70%	92%

In Table 4, we show another example for another SMG2000 scenario. In this scenario, there are 29 valid communication patterns. Using the Non-contextual approach, we were able to detect 20 valid patterns. When applying our contextual approach, we were able to detect 27 valid communication patterns.

As the process topology becomes complex, the precision of our approach decreases. The recall seems to stay relatively the same, which is a good thing. At least, we are not missing many valid patterns. When we analyzed the traces, we noticed that the low precision was due to the inability for our pattern detection algorithm to deal with nesting patterns. Consider the following example:

Function 2: R1R2R3
Function 2: S1S2S3R1R2R3S1S2S3R1R2R3

The algorithm will detect both R1R2R3 and S1S2S3R1R2R3 as patterns, whereas the valid pattern in this case is only S1S2S3R1R2R3. Therefore, we need to further fine-tune the algorithm to deal with nesting patterns.

TABLE 4. NUMBER OF DETECTED COMMUNICATION PATTERNS FOR SMG2000 TRACE (4X4X1 PROCESS TOPOLOGY; 4X4X4 INPUT)

Pattern Detection Technique	Patterns	Precision	Recall
Non-Contextual	78	25%	69%
Contextual	42	64%	93%

In summary, we have clearly demonstrated that the program functions, used as a context, can improve the automatic detection of valid inter-process communication patterns. The approach achieves a reasonably good precision and recall. But clearly, we need to continue to experiments with larger systems for identifying opportunities to improve both measures.

C. *Performance Evaluation*

We also compared the performance of the two approaches in terms of processing time. In the following, we present the statistics from several process topologies from both Sweep3D and SMG2000.

Our experiments are conducted on a 1.83 GHz Intel Core 2 Duo CPU with 3.0 GB of RAM. It is worth mentioning that the pattern detection algorithm runs on each process trace separately (sequentially) and the execution times shown in this study are the sum of the execution times of all runs. Conducting the experiments in a parallel setting is expected to significantly improve the execution time.

Sweep3D: We used different process topologies to vary the number of processes. As we can see in Table 5, the performance of the contextual approach is slightly lower than a non-contextual approach. This is expected since the contextual approach uses additional information (i.e., function call trees) which affects the processing time. The improvement we made to the performance of the previous algorithm, however, seems to bring its fruits since the performance gap between the two approaches is very small in all cases (average difference of around 3 seconds).

SMG2000: Similar to Sweep3D, we did not find any significant difference in terms of performance between the contextual and the non-contextual approach as shown in Table 6.

TABLE 5. PERFORMANCE ANALYSIS FOR SWEEP3D TRACES

Process Topology	It.	Messages	Contextual (seconds)	Non-contextual (seconds)
2 x 3	12	20160	0.78	0.72
6 x 3	12	51840	2.45	1.674
5 x 5	40	256000	5.83	4.20
7 x 4	74	532800	9.156	7.20
8 x 8	120	2150400	28.74	22.27
8 x 16	120	4454400	56.45	52.23
		Average	17.24	14.71

TABLE 6. PERFORMANCE ANALYSIS FOR SMG2000 TRACES

Topology	Problem Size	MPI Events	Contextual (seconds)	Non-Contextual (seconds)
8x1x1	2x2x2	9312	1.25	0.98
2x2x2	2x2x2	25416	1.33	1.40
4x4x2.	2x2x2	248768	12.56	10.82
16x1x1	10x10x10	978296	73.98	68.71
32x1x1.	10x10x10	2363156	162.14	147.65
64x1x1.	10x10x10	5324304	359.54	354.32
		Average	101.8	97.31

V. THREATS TO VALIDITY

A threat to the validity of our conclusions exists because we have only experimented with two systems. Though we

agree that more experiments are needed, the systems used in this study are benchmark applications used also by other researchers [22, 23, 32].

A threat to construct validity exists in the way we validated the results. We used system documentation to validate the detected patterns. We also looked at the source code of the target systems to understand the supported communication patterns. An ideal validation would be to conduct a formal experiment with software engineers. However, it was challenging to find HPC systems in which we had access to the developers. We mitigated this threat by choosing open source systems that are benchmark applications in the field, with adequate documentation.

A threat to internal validity exists in the implementation of the programs to process trace information and to detect patterns. This is because we automated this procedure by writing shell scripts and Java code. We have minimized this threat by manually investigating the outputs, and making sure our results are valid and consistent.

VI. RELATED WORK

Trace analysis has been the topic of many studies (e.g. [4, 11]) in the area of reverse engineering and program comprehension. Much of the research, however, focuses on single-threaded systems paying less attention to traces of distributed systems. The analysis of the dynamics of HPC systems, in particular, is still in its infancy. In this section, we report on the key studies that are related to this paper.

Preissl et al. [22, 23] proposed an algorithm for the detection of inter-process communication patterns in MPI traces. Their approach was based on compressed suffix trees. They used MPI seed events and static analysis to determine areas in the code where communication patterns could occur. Using this approach, the authors were able to show how communication patterns, once detected properly, could be used to improve the program performance. Using static analysis, however, poses another set of challenges because of the parallel nature of HPC systems. In addition, static analysis requires building and storing a static model of the system, which adds complexity to the analysis technique. The main difference between the work presented in this paper and theirs is that we use a pure dynamic analysis technique, which we believe is more efficient and practical.

Wolf et al. [32] used knowledge from virtual topologies to identify patterns of inefficient behaviour due to long wait states caused from inefficient application of the parallel programming model. The communication topology is used to identify the phases of inter-process communication in the program. This work is different from our work since it only looks for patterns of inefficient behaviour resulting from processes in long wait states and it does not aim at recovering inter-process communication patterns. Also, the authors assume that knowledge of the communication topology is available, which is not always the case.

Knüpfer et al. [14] proposed an algorithm to remove contiguous repeating patterns from HPC traces to reduce the size of the trace. The algorithm is based on the compressed complete call graph (cCCG) and the pattern graph (a derivative of the cCCG). An advantage of using cCCG is that it references all call sequences that are equal with respect to a call structure and temporal behaviour, which improves trace compression. In their algorithm, they only detect contiguous pattern repetitions. This approach does not focus on detecting communication patterns. It only detects repeating patterns on each process trace separately represented by the different types of events collected in the trace.

Kunz and Seuren [8] presented a technique based on finite state automata to find communication patterns in the trace that match an input pattern. The pattern matching algorithm is performed by determining the longest process pattern in the input communication pattern which will be used as the search string in the pattern matching algorithm. They start building the communication pattern by locating the partner events on the other process traces. This approach is only concerned with detecting patterns based on a predefined input pattern. In this paper, we propose an approach that finds repeating communication behaviour without prior knowledge of the type of communication used in the program. In our previous work, we presented an n-gram based approach that is used for locating a predefined communication pattern in an MPI trace [1].

Köckerbauer et al. [15] proposed the use of a pattern matching technique to simplify the debugging of large message passing parallel programs. This achieved this by identifying patterns in the trace file that are similar to a predefined pattern. First, the user specifies a description of the communication pattern to be searched for in the trace file. This pattern description is then translated to abstract syntax trees. The ASTs are then scaled up to the number of processes in the trace (or the number of the target processes in the trace). The pattern matching process is run on each process trace individually. In their work, they used a hash-based search to detect exact and similar patterns on each process trace. Finally, the matching patterns are merged to get the communication pattern which should be exact or a variation of the user's specified pattern. In this work, we focus on detecting communication patterns in an MPI trace without prior knowledge of the communication patterns used in the program.

Moore et al. [19] proposed a pattern matching method for detecting patterns of inefficient behaviour based on wait states in order to be used in KOJAK (a performance analysis tool for high performance parallel applications) [31]. These patterns of inefficient behaviour are identified by converting the trace into a compact call-path profile which classifies patterns based on the time spent in communication. This approach only looks for events that cause performance degradation and does not focus on inter-process communication patterns.

VII. CONCLUSIONS AND FUTURE WORK

We developed a contextual-based approach for detecting inter-process communication patterns from HPC traces. We applied this method to traces of two systems to test its effectiveness. We achieved a precision and recall superior to a

simple non-contextual method. The key finding of this study is that function calls serve as a context to guide the pattern extraction process.

To build on this work, we need to gain more comprehensive knowledge of the variables defining a communication context in HPC systems. This would help us design metrics of cohesiveness that would ultimately improve the detection precision and recall. We also need to better understand the relationship among patterns, especially for large topologies.

ACKNOWLEDGMENT

This work is supported partly by NSERC (Natural Sciences and Engineering Research Council of Canada).

REFERENCES

[1] L. Alawneh and A. Hamou-Lhadj, “Pattern Recognition Techniques Applied to the Abstraction of Traces of Inter-Process Communication,” *In Proc. of the European Conf. on Software Maintenance and Reengineering,* pp. 111-120, 2011.

[2] L. Alawneh, A. Hamou-Lhadj, “Identifying Computational Phases from Inter-Process Communication Traces of HPC Applications,” *In Proc. of the 20th International Conference on Program Comprehension (ICPC),* pp. 133-142, 2012.

[3] A. Chan, W. Gropp, and E. Lusk, “An efficient format for nearly constant-time access to arbitrary time intervals in large trace files,” *Journal of Scientific Programming, 16(2-3),* pp. 155–165, 2008.

[4] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, R. Koschke, “A Systematic Survey of Program Comprehension through Dynamic Analysis,” *IEEE Transactions on Software Engineering (TSE), 35(5),* pp. 684-702, 2009.

[5] M. Desnoyers and M. R. Dagenais, "Tracing for Hardware, Driver, and Binary Reverse Engineering in Linux,” *Code Breakers Journal, 1(2),* 2006.

[6] J. P. Downey, R. Sethi, and R. E. Tarjan, “Variations on the Common Subexpression Problem,” *Journal of the ACM, 27(4),* pp. 758-771, 1980.

[7] K. El Maghraoui, B. K. Szymanski, and C. A. Varela, “An Architecture for Reconfigurable Iterative MPI Applications in Dynamic Environments,” *In Proc. of the 6th International Conference on Parallel Processing and Applied Mathematics, number 3911 in LNCS,* pp. 258-271, 2005.

[8] T. Kunz and M. F. H. Seuren, "Fast detection of communication patterns in distributed executions,” *In Proc. of the Conference of the Centre for Advanced Studies on Collaborative research (CASCON),* 1997.

[9] M. Geimer, F. Wolf, B. J. N. Wylie and B. Mohr, “Scalable parallel trace-based performance analysis,” *In Proc. of the 13th European PVM/MPI Users’ Group Meeting, vol. 4192 of LNCS,* pp. 303–312, 2006.

[10] A. Hamou-Lhadj, "Techniques to simplify the analysis of execution traces for program comprehension," *PhD thesis dissertation, University of Ottawa,* 2005

[11] A. Hamou-Lhadj A. and T. Lethbridge, "An Efficient Algorithm for Detecting Patterns in Traces of Procedure Calls," *In Proc. of the 1st ICSE Workshop on Dynamic Analysis (WODA),* 2003.

[12] A. Hamou-Lhadj and T. Lethbridge, “An Efficient Algorithm for Detecting Patterns in Traces of Procedure Calls,” *In Proc. of the 1st ICSE International Workshop on Dynamic Analysis,* pp. 33-36, 2003.

[13] R. Karp, R. E. Miller, A. L. Rosenberg, “Rapid Identification of Repeated Patterns in Strings, Trees and Arrays,” *In Proc. of 4th Symposium of Theory of Computing,* pp.125-136, 1972.

[14] A. Knüpfer, B. Voigt, W.E. Nagel, and H. Mix, “Visualization of repetitive patterns in event traces,” *In Proc. of the Workshop on State-of-the-Art in Scientific and Parallel Computing,* 2006.

[15] T. Köckerbauer, T. Klausecker and D. Kranzlmüller, “Scalable Parallel Debugging with g-Eclipse,” *Book Chapter, Springer Berlin Heidelberg,* 2010.

[16] D. Kranzlmüller, “Event Graph Analysis for Debugging Massively Parallel Programs,” *Ph.D. Dissertation, GUP Linz, Johannes Kepler University* Linz, Austria, 2000.

[17] C. Ma, Y. M. Teo, V. March, N. Xiong, I. R. Pop, Y. X. He, and S. See, “An approach for matching communication patterns in parallel applications,” *In Proc. of the International Symp. on Parallel & Distributed Processing,* pp.1-12, 2009.

[18] MPI: A Message Passing Interface Standard, June 1995. URL: http://www.mpi-forum.org.

[19] S. Moore, F. Wolf, J. Dongarra, S. Shende, A. Malony, and B. Mohr. “A Scalable Approach to MPI Application Performance Analysis,” *In Proc. of the 12th European Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface, LNCS, 3666,* pp. 309–316, 2005.

[20] OpenMP URL: http://openmp.org/wp/

[21] N. Palma, “Performance Evaluation of Interconnection Networks using Simulation: Tools and Case Studies,” *Ph.D. Dissertation, Department of Computer Architecture and Technology,* University, Spain, 2009.

[22] R. Preissl, T. Köckerbauer, M. Schulz, D. Kranzlmüller, B. R. de Supinski, and D. J. Quinlan, “Detecting patterns in MPI communication traces,” *In Proc. of the 37th International Conference on Parallel Processing,* pp. 230–237, 2008.

[23] R. Preissl, B. R. de Supinski, M. Schulz, D. J. Quinlan, D. Kranzlmüller, T. Panas, “Exploitation of Dynamic Communication Patterns through Static Analysis,” *In Proc. of International Conf. on Parallel Processing,* pp. 51-60, 2010.

[24] K. Sadakane. “Compressed suffix trees with full functionality,” *Theory of Computing Systems,* 2007.

[25] SMG200 Banchmark Code, URL: http//www.llnl.gov/asc/purple/benchmarks/limited/smg/

[26] Sweep3D, Accelerated strategic computing initiative. The ASCI Sweep3D Benchmark Code. URL: http://public.lanl.gov/hjw/CODES/SWEEP3D/sweep3d.html, LANL 1995.

[27] L. Uribe, P. Costanza, T. D'Hondt, "Adding data-movement constructs to the PGAS parallel computing model," *In Proc. of the 7th ACM European Lisp Workshop,* 2010.

[28] VampireTrace (The Vampir Performance Visualizer) Tool, Available Online, URL: http://www.vampir.eu

[29] T. A. Welch, “A technique for high-performance data compression,” *IEEE Journal of Computer, 17(6),* pp. 8-19, 1984.

[30] F. Wolf, D. Becker, M. Geimer, and B. J. N. Wylie, “Scalable performance analysis methods for the next generation of supercomputers,” *In Proc. of the John von Neumann Institute for Computing (NIC) Symposium,* pp. 315 – 322, 2008.

[31] F. Wolf and B. Mohr. “KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Applications,” *In Proc. of the European Conference on Parallel Computing (Euro-Par), volume 2790, LNCS,* pp. 1301–1304, 2003.

[32] F. Wolf, B. Mohr, J. Dongarra and S. Moore, “Automatic Search for Patterns of Inefficient Behavior in Parallel Applications,” *Wiley Journal on Concurrency Practice and Experience,* pp. 1481 - 1496, 2006.